Advanced Programming for the 16K ZX81

MIKE COSTELLO

A large number of ZX81 users have now had time to get used to the machine and are looking for more information in order to exploit it to its full potential. This book is written for such readers.

Drawing on his experience with other microcomputers, the author describes techniques that can be applied to the ZX81 in order to overcome some of its inherent limitations. This involves some investigation of the ZX81's operating system, discussion of BASIC subroutines and techniques useful in a wide range of programs, including business applications and games, as well as details of the application of Artificial Intelligence techniques to programming for the ZX81.

Later chapters in the book are devoted to the use of assembly language programming techniques, hybrid programming — mixing BASIC with machine code, and developing utility programs to suit the user's own particular needs.

Mike Costello became involved with microcomputers in 1979, working initially on a TRS-80. He has published a number of programs for this machine. More recently he has been studying the ZX81 as a low-cost means of spreading computing skills, and has also published programs for this machine, as well as articles on microcomputing. He now runs his own software and publishing business and edits *The War Machine* — a magazine on computer simulation games.

ISBN 0 333 34590 8

Advanced Programming for the 16K ZX81

Macmillan Computing Books

Advanced Graphics with the Sinclair ZX Spectrum I.O. Angell and B.J. Jones

Assembly Language Programming for the BBC Microcomputer Ian Birnbaum

Advanced Programming for the 16K ZX81 Mike Costello

Microprocessors and Microcomputers — their use and programming Eric Huggins

The Alien, Numbereater, and Other Programs for Personal Computers –with notes on how they were written John Race

Beginning BASIC Peter Gosling

Continuing BASIC Peter Gosling

Program Your Microcomputer in BASIC Peter Gosling

Practical BASIC Programming Peter Gosling

The Sinclair ZX81 – Programming for Real Applications Randle Hurley

More Real Applications for the Spectrum and ZX81 Randle Hurley

Assembly Language Assembled – for the Sinclair ZX81 Antony Woods

Digital Techniques Noel Morris

Microprocessor and Microcomputer Technology Noel Morris

Understanding Microprocessors B. S. Walker

Codes for Computers and Microprocessors P. Gosling and Q. Laarhoven

Z80 Assembly Language Programming for Students Roger Hutty

Advanced Programming for the 16K ZX81

Mike Costello

First published 1983 by
THE MACMILLAN PRESS Ltd
London and Basingstoke
Companies and representatives
throughout the world

Printed in Great Britain by Unwin Brothers Limited,
The Gresham Press, Old Woking, Surrey.

ISBN 0 333 34590 8

Cover photograph of Sinclair equipment kindly supplied by Sinclair Research Limited.

Contents

Preface

This book is intended for ZX81 owners who have become familiar with the use of BASIC on the machine and are interested in developing their programming techniques. It is also aimed at those who have looked at the Z80 instruction set, and have thought about writing some programs in assembly language, but are not clear how to get started on this particular computer.

The next six chapters, on BASIC programming, will illustrate some of the potential of the ZX81 as well as pointing out its limitations. The experience gained by users of more powerful computers can often be applied to the ZX81 to speed up program development and improve the quality of finished programs. The final three chapters are for the assembly-language programmer and deal both with stand-alone machine-code programs and with machine-code routines which are designed to interface with BASIC programs.

These final three chapters include details of ROM calls that can be used by any assembly-language programmer, and which speed up programming by providing ready-made routines to handle such matters as output to the TV display and reading data in from tape. These calls are then incorporated in assembler routines which the programmer is likely to need in the course of developing his own programs. Each such routine is explained step by step, and an explanation is also given of the reason for using machine code instead of BASIC for that particular program. However, these chapters do not teach Z80 assembler from scratch, and some knowledge of the Z80 instruction set on the part of the reader is assumed.

Although many of the shorter subroutines included in the book would fit within 1K, it will be assumed that the reader is using a 16K version of the machine, as one cannot get very far without it. A ZX81 fitted with a 16K pack represents a potentially very powerful computer at an astonishingly low price.

Mention should be made of the different ROMs fitted to ZX81's since the machine was launched. So far as is known, there are only two versions of the ROM; the first was included in the earliest ZX81's that were manufactured and contained a couple of errors. The first error is responsible for the advice given on page 127 of the Sinclair manual, that a value must be POKEd immediately after using PAUSE; the second error caused unpredictable results when using the ZX81's floating-point arithmetic for certain types of

calculation. The second version of the ROM clears up these errors, but as a result, some of the machine-code routines that make up the ROM now start at slightly different addresses. This affects the last three chapters of the present book, and addresses for both versions of the ROM are given in these chapters where necessary.

A note on conventions: the practice throughout the book has been to use capital letters for words which are BASIC keywords or which are Z80 assembler opcodes, to distinguish them from words that are not being used in these senses. The word "Listing" will appear with a capital letter when it refers to a program listing printed in this book, but will appear as "listing" when it refers to a BASIC program in the memory of the ZX81.

In the chapters on assembly language, names such as SLOW and FAST will be used to refer to various ROM routine addresses. These are just mnemonic labels that I have invented to make it easier to keep track of what is happening in an assembly-language Listing, and have no particular authority.

Acknowledgement should be made to the work of Dr Ian Logan, as the starting addresses of several of the ROM routines used in this book were first identified in his book *Understanding Your ZX81 ROM* (Melbourne House, 1981).

At the time of writing, a great deal of software has appeared on the market for the ZX81, and there has been time for programmers to settle down with the machine, learn its peculiarities and start producing high-quality programs. I think most observers will agree that the overall quality of the software that is currently available does not stand up at all well to comparison with (for example) the range and quality of software that had become available for computers such as the TRS-80 and PET, when these machines had been on the market for an equivalent length of time. Whatever the reasons for this, it would be a good idea for British programmers to make a somewhat better showing in this regard than they have done to date. The ZX81 is now assuming the status of the universal entry-level machine for those who have not previously experienced personal computing, and they will get a strange idea of what can be done with a microcomputer if they form their assessment on the basis of the currently available software. Now that the machine is penetrating deeply into overseas markets, new programs will be generated by users in other countries. It would be somewhat mortifying if this software, imported into this country, were to assume dominance over native products. The general intention of the present book is to encourage a greater level of sophistication in ZX81 programming; the machine presents a number of obstacles to methodical program development, but intelligent programs can still be written for it if appropriate techniques are used.

Chapter 1 Saving space, speeding up and keeping things clear

The ZX81 presents a number of problems to the programmer who wishes to write BASIC programs longer than two or three K. Execution speed is very poor, far poorer than the 3 MHz clock rate would indicate, owing to a number of design features of the machine. Certain statements take up an inordinate amount of memory and even a moderately ambitious program runs the risk of filling 16K memory, thus requiring the programmer to stop and think up ways of compacting his coding in order to allow him to complete his project. When the program is complete and a copy is dumped to the printer, a great deal of additional documentation may have to be added if the programmer wants to be sure that he will be able to understand its workings at a later date. The purpose of this chapter is to give advice on these three topics so that programs can be designed from scratch to minimise such problems.

SAVING SPACE

Let us first look at the problem of excessive memory consumption. A great deal has been written on this subject but it does not take very long to summarise the various methods by which BASIC programs can be compacted to occupy the minimum amount of memory. It is, however, necessary to be clear on the way in which the ZX81 stores program lines and variables, or one may find oneself using some elaborate programming technique to save space which, on further examination, turns out to offer no improvement in net consumption of RAM.

There is a five-byte overhead for storing any BASIC program line. We know that all programs start at location 16509, so we can say that the line number of the first line of a program is at locations 16509 and 16510, held as a two-byte value composed of a Less Significant Byte and a More Significant Byte. Unusually, however, the MSB is stored first and followed by the LSB. In practice the values stored here will never be outside the range 1-9999 decimal, the line range permitted by the interpreter.

The next two bytes contain the length of the program line including the end-of-line marker. This time the standard sequence of LSB followed by MSB is used, and there appears to be no practical limitation on the maximum program line length. The line-length value is of course used by the interpreter when hunting through the program listing for a line number corresponding to a GOTO or GOSUB reference, as it allows it to leapfrog from the start of one line to the start of the next without actually reading the text of each.

The program line itself is stored in a fairly conventional manner with tokens used to represent BASIC keywords, as in Microsoft BASIC interpreters. However, all numeric literals are, notoriously, stored in a different way; the value itself is stored as a sequence of byte values representing the digits of the number in the Sinclair character codes, and then comes the value 126 which means, in effect, "floating-point follows". The next five bytes hold the floating-point representation of the number. When the program is run, these five bytes are copied to the variable table as the line in question is executed.

The idea is, of course, that the processing time needed to convert from ASCII to floating-point should not slow down execution of the program, since the conversion is done at the time that the line is entered into memory during editing or when the program is first typed in. The pity of it is that the great majority of numeric literals in any program are integers, used for loop counters, screen positions for PRINT AT statements and so forth. The lack of any routines for handling two-byte integer values in programs makes the Sinclair ROM much slower in the execution of most programs than would otherwise be the case, and increases the length of lines considerably.

At the end of each BASIC line comes the value 118. A second 118 follows the last line of a program; it is the address of this byte in RAM that is contained in locations 16396/7 as the start of the Display File.

When trying to compact programs, we will have to give our attention first of all to the problem of numeric literals. Next in order of priority is the trouble caused by the ZX81's inability to handle multi-statement lines, as the five-byte overhead for starting a new line must be paid however short the line is. Then there is the problem of storage of string literals, which may require several thousand bytes of RAM in a typical program. Let's look at this problem first, then return to the other two topics.

In many computers, such as the TRS-80, a string variable is stored in the program listing itself. For example, suppose you have the program:

```
10      LET A$="STRING"
```

When this program is run, the TRS-80 will set up storage in its variable table for the variable A$, together with information about its value. However, it will not store the six characters of the string itself in the table; all it stores is the length of the string (in one byte) and the address *within the program listing* of the first byte of the string (in two bytes). All such variables therefore take up three bytes (plus three more for associated values and markers), irrespective of the length of the string, although obviously the length of the string affects the program listing itself.

In the ZX81, of course, running line 10 will cause the interpreter to set up A$ within its variable table - three bytes for the name of the variable followed by its length, and then six bytes for the text of the string. A$ is now duplicated in two places within memory and this sort of situation is to be avoided whenever possible.

What the programmer can always do is delete all the lines which set up the initial values of numeric and string variables, after running the program. Now all the values are in the variable table and nowhere else, and are preserved thanks to the machine's ability to maintain its current variable table while the total length of the BASIC program changes due to editing.

Unfortunately, efficient program development is very difficult when the text of the program listing no longer displays the initial values of all the variables. Although it may be feasible to delete all these initialising lines at the end of the process of developing the program, some compromise technique is likely to be necessary at an intermediate stage, as memory fills up but the programmer still needs to be reminded of the initial variable values. One way out of this relies on the fact that tokenised keywords in program lines remain tokenised when the line is converted to a REM. Suppose you type in:

```
10      LET R=4
```

This line occupies 15 bytes, including the five-byte overhead for setting-up the line. Here are details of the values held in memory which represent the above line:

```
Line-number    Length    LET token     R    =     4
  0  10         11  0        241       55   20    32

Floating-point follows    Floating-point version of 4
          126                  131  0  0  0  0

End of line
   118
```

Now run the program to set up "4" in the variable table. Next edit the line so that it reads:

```
10      REM LET R=4
```

The total memory consumption of the line has now been reduced to 10 bytes, taken up as follows:

```
Line-number    Length    REM token    LET token    R    =    4
  0  10          6  0       234          241       55   20   32

End of line
   118
```

The six bytes of single-precision variable storage have disappeared into oblivion; the interpreter does not need to store them because a REM line cannot be executed, but is just skipped when a program is running.

You can keep the line like this until the program is complete, and then delete it; alternatively, you can quickly delete the REM and restore the line to its original form. Bear in mind, however, that the procedure doesn't work the other way round. That is, if you begin by typing:

```
10    REM L
```

the interpreter will not allow LET to be typed as a one-byte keyword and will insist that the letters of the keyword be typed in one at a time, taking up one byte of storage for each letter.

In cases where the line must be left in the final program, or where the value to be stored is not an integer such as 4, the general rule is to use VAL. The line:

```
10    LET X=9
```

takes 15 bytes, whereas the alternative version:

```
10    LET X=VAL "9"
```

only takes 12. The same three-byte saving can be made for larger numbers. However, a greater saving can be made if the number is an integer less than 256. Suppose the number is 249, which happens to be the keyword-code for RAND. Start with:

```
10    RAND
```

Then edit it to insert some additional material, so that you get:

```
10    LET X=CODE " RAND "
```

The spaces that appear on either side of RAND do not represent extra bytes of memory in the program listing, but are just a peculiarity of that part of the interpreter that adds spaces to listings of lines before printing them on the screen. In fact the above line takes only 12 bytes, compared to:

```
10    LET X=249
```

which would take 17.

Here is the way that each of these versions of the line are held in memory:

```
10    LET X=CODE " RAND "

Line-number   Length   LET token   X   =   CODE token
  0  10         8  0      241      61  20       196

"    RAND token   "    End of line
11       249      11       118


10    LET X=249

Line-number   Length   LET token    X   =   2   4   9
  0  10         13  0      241      61  20  30  32  37

Floating-point follows   Floating-point version of 249
         126               136  121  0  0  0

End of line
    118
```

The same comment applies about the sequence of typing that is necessary to enter the line to memory. If you start with:

```
10    LET X=CODE "RAND"
```

the line is accepted, but RAND occupies four bytes. In any case, you will discover that X is actually set equal to CODE "R".

You may be wondering how this technique can be applied to other one-byte integer values, in cases where the value doesn't correspond to a BASIC keyword (or graphics character) that can be entered directly from the keyboard. It's still possible to get the value into one byte within the program line, but a temporary line must be inserted into the BASIC program to POKE the value. Listing 1.1 illustrates how this is done. The idea in this example Listing is

Listing 1.1

```
  50 LLIST
  90 POKE (PEEK 16425+(PEEK 1642
6)*256)+9,93
 100 LET C=CODE "*"
 120 LPRINT ,,
 130 LLIST

  50 LLIST
  90 POKE (PEEK 16425+(PEEK 1642
6)*256)+9,93
 100 LET C=CODE "?"
 120 LPRINT ,,
 130 LLIST
```

to get the value 93 into line 100. The first LLIST of the program shows the original BASIC program; the second LLIST shows the altered program after it has RUN. In other words we are using self-modifying code in BASIC, and it's interesting to see how straightforward it is to apply this technique, at any rate on a minor scale.

The routine works because the values stored at 16425/6 represent the address in memory of the start of the next BASIC line of the program. These values are stored in normal LSB/MSB format, so line 90 extracts the decimal equivalent of the address and adds 9 to it. If you look at line 100 you will see that the asterisk is 9 bytes from the beginning of the line; if you are not clear about this, consult the Sinclair manual which explains that two bytes are used to store the number of a line, another two to hold the length, one byte for each BASIC keyword and so on. When the interpreter encounters line 90, the values stored at 16425/6 represent the address of the next line (100), so the POKE puts the value 93 into the memory location that previously held the byte value representing an asterisk. The interpreter doesn't know how to handle this value when the line is LLISTed and just prints a query, but you can PRINT C to convince yourself that the technique works. Line 90 can now be deleted. There are any number of applications for this byte-storage method, some of which will be mentioned later in the book.

It's often said that small integer values should be set up using Boolean operations and function-keywords like PI. This is certainly possible, although it can be tricky to arrive at values like 7 or 9 without taking several lines of programming to get there, and this is a bit pointless because of the five-byte overhead for starting a new line. The following little routine may be of use if you have a long program which needs a great number of small integers of this kind:

```
10    LET Z=NOT PI
20    LET O=Z=Z
30    LET T=O+O
40    LET F=T+T+O
```

Further values can be derived from combinations of the values zero, one, two and five produced by this initial routine, rather like making up an amount of pence with a combination of copper coins. This line, for example, slices the first five characters out of A$ and transfers their numeric equivalent to X:

```
100   LET X=VAL A$ (O TO F)
```

This shows a way of storing large values in strings consisting of sequences of decimal digits, and then extracting them in chunks using the TO slicer. Unfortunately, A$ will duplicate itself in the variable table as soon as the program is run. The real answer to this problem is to store data in REMs within the program, and extract it by "reading" the program listing itself as it resides in memory. Although this can be done using PEEK, this is a very slow and restrictive approach; it's much better to use a short machine-code routine to read through the listing until it reaches a marker byte that precedes the data to be collected. Later in the book, a technique for achieving this will be explained.

As a final word on the subject of numeric value storage, one should not forget exponential notation and expression evaluation. The first allows the odd byte to be saved here and there, as in:

```
100   GOTO 3E3        rather than     100    GOTO 3000
```

Exponential notation can also be used for PAUSEs, and combined with VAL, as in:

```
100   PAUSE VAL "4E4"
```

Expression evaluation refers to the ability of the interpreter to calculate the numerical equivalent of an arithmetic expression without the programmer needing to type in a separate LET statement. We can say, for example:

```
10    LET A=VAL "11**4"
```

and save one byte compared to:

```
10    LET A=VAL "14641"
```

Opportunities for saving space this way are obviously very limited!

Earlier in the chapter I mentioned the problem of multi-statement lines, which are not permitted by the interpreter. It is often argued that programmers should avoid using multi-statement lines anyway, in the interests of clarity. However, this can increase the length of a program considerably, particularly when the program needs to carry out a long series of tests of the values of many variables, in order to determine the vector for a conditional jump. There might for example be ten different subroutines, accessed through GOSUB, and the program will call one of them after evaluating half-a-dozen variable values. Programs with this sort of structure occur frequently in Artificial Intelligence work. How does one code a ZX81 program to achieve this sort of result?

Typically, the programmer needs to test the values of, shall we say, three variables: A, which has the value 1, 2, 3 or 4; B, with the same value range; and C, which has an integer value between zero and 100. A particular case can be tested like this:

```
100   IF A=3 AND B=2 AND C<10 THEN GOTO 2000
```

The trouble is that a new line must be typed in for every unique case. To avoid this, the programmer can use a single line that tests for every possible case:

```
100   GOTO (2000 AND A=3 AND B=2 AND C<10)+(2100 AND A=4 AND
B=2 AND C<10)+...
```

However, an enormously long line will be needed to cover all the permutations of possible outcomes of variable values, and there is obviously a lot of duplication in such a line, as cases like B=2 will be quoted again and again in combination with other variable values. One alternative is to combine the above procedure with IF-THEN statements, as in:

```
100   IF A=3 THEN GOTO (2000 AND B=2 AND C<10)+(3000 AND B=2
AND C>9)+...
```

One can even nest the IF-THENs, like this:

```
100   IF A=3 THEN IF B=2 THEN GOTO (2000 AND C<10)+(4000 AND
C>9)+...
```

Although this looks like a way out, it still turns out to be necessary to code a large number of lines to cover all cases. The nested IF-THEN structure is like a series of sieves, each one cutting a range of possibilities out of consideration and concentrating on a single possibility. A new line must then be started if one wants to go back to considering a possibility that was cut out earlier, such as A=1. The reason for this, of course, is that the ZX81 does not permit the ELSE statement, which would allow the programmer to "de-nest" the logic of the line and come back to cover all the remaining possibilities.

One solution that is sometimes offered is to use the rather unusual ability of ZX81 BASIC to accept evaluated expressions as line numbers for GOTO and GOSUB. In the example so far considered, we could say:

```
100   GOTO 1000+A*100+B*10
```

Now we can cover all the possibilities with an orderly sequence of lines; lines 1110, 1120, 1130 and 1140 will deal with values of B from 1 to 4 respectively, with A=1. Lines 1210, 1220, 1230 and 1240 will do the same for A=2, and so on. This solution is in fact the best one to adopt when we want something different to happen for every unique case; we would in any case need a set of program lines to describe what happens in each such case, and line 100 will send control to the first line of the appropriate set. But this procedure will be very wasteful of memory if there is redundancy in the range of possibilities being considered, that is, if there are only (say) seven different outcomes from the 16 different permutations of A and B. We will then find ourselves creating a subroutine to handle each of the seven outcomes, and sending control to the appropriate subroutine from a number of places within the program. For example, if the subroutine at 8000 is to be used when A=1 and B=2, and also when A=2 and B=1, we will have:

```
1120  GOTO 8000
...
1210  GOTO 8000
...
```

and so on each time the values of A and B are such that the subroutine at 8000 represents the action to be taken.

When faced with this sort of problem, a programmer will usually search for a "default condition" which is to apply when all other conditions have been tested for and rejected. We would code a single long line covering the permutations that do *not* involve passing control to the subroutine at 8000:

```
100   GOTO (1110 AND A=1 AND B=1)+(1130 AND A=1 AND B=3)+...
```

Then we would expect that, if none of the cases meets the condition, control would drop through to the next line and we could say:

```
110   GOTO 8000
```

Unfortunately, ZX81 BASIC doesn't work this way. If none of the cases quoted in line 100 meets the current condition, the computer will simply go into an internal loop, cycling around endlessly looking for a value line-number to jump to. Fortunately one can escape from the loop by hitting BREAK, whereupon one is given a 0 report code and some line number unrelated to the line that has caused the trouble; this is not much consolation, and this undocumented bug in the interpreter makes lines of this type much less useful to the programmer than they would otherwise be.

It turns out that there is an elegant solution to this kind of problem, which is also extremely economical of memory. In order to make the exposition of the method entirely clear, it's necessary to start with a real-life programming task which requires such methods to be used. This will be done in Chapter 6, where we will show the use of the technique in an Adventure program. For the time being, however, the following example program shows the essential idea:

```
10    DIM V$(4,4)
20    LET Z=8000
30    LET V$(1,2)="Z"
100   GOTO VAL V$(1,2)
200   STOP
8000  PRINT "GOT HERE"
```

Although the Sinclair manual gives little indication that ZX81 BASIC can be used in this way, the program in fact runs perfectly; the first three lines can then of course be deleted, and the program will still operate if you GOTO 100. The string array occupies a total of 24 bytes in the variable table; each line number which may be the object of a GOTO needs 6 bytes in the variable table, given that it is stored as a single-letter numeric variable. In Chapter 7 we will see how to calculate the memory consumption of such a program in advance, so that the programmer can be confident of having enough memory left to finish the project, and how to

document the contents of arrays of this kind during program development.

SPEEDING UP

The first thing that becomes apparent when we consider the opportunities for speeding-up BASIC programs is that they are in conflict with most of the methods, just described, for compacting program listings. There is generally a strict trade-off in the sense that increased execution speed will be at the expense of memory saving, and vice versa. VAL is an example of this; the interpreter will take much longer to convert, for example, VAL "123" to the floating-point representation of the number 123 than it will take to read the floating-point equivalent of 123 off the program line, where it is already stored, and copy it to the variable table. As a rule of thumb, expect a twofold increase in the time taken for the interpreter to deal with a number expressed by VAL, and then add another twofold delay for each character that makes up the value. For example, VAL "123" will cause a delay of $\times 2$, plus $\times(2 \times 3)$ for the three characters making up the value, giving a total eightfold delay in executing the program line.

The technique of expressing values as the sum of other values, as used in the short routine for generating Z, O, T and F, will also slow things down, and delays can be expected when using any technique for storing data in compacted form within the program listing and then extracting it again. However, a distinction should be drawn here between extracting a known value from a known location on the one hand, and searching for a given value on the other. The first technique is exemplified by the following lines:

```
10    LET A$="GO*GETDROJUMQUISAV"
...
1000  IF Z$=A$(7 TO 9) THEN GOSUB DROPROUTINE
```

This is part of a vocabulary-search routine as it might be used in a simple adventure game. The programmer has arranged that the first of the words input by the player, which will always be some kind of verb if two-word commands are being used, is collected in Z$, any excess of characters after the first three being chopped off and an asterisk added in the case of any two-letter words. In line 1000, the programmer only wants to test whether the first word was DROP; he will have to add further lines, each testing explicitly for one of the permitted words (go, get, jump, quit game and save game). Any memory saving resulting from this technique will become apparent when the programmer applies it to the list of second words (nouns), as there will typically be many more of these and their length will tend to be greater. The routine will, however, execute very quickly, because the TO slicer does not create a copy of the string being manipulated as part of the procedure for manipulating string expressions. The long delays experienced by users of machines such as the TRS-80 and PET

while string manipulation is carried out can generally be avoided on the ZX81.

By contrast, the following routine will probably turn out to be too slow for practical use:

```
10    LET AS="GO*GETDROJUMQUISAV"
...
1000  FOR X=VAL "1" TO VAL "16" STEP VAL "3"
1010  IF Z$=A$(X TO X+2) THEN GOTO 2000+X*100
1020  NEXT X
```

Here the programmer is using a computed GOTO to jump to the routine that handles the particular verb input by the player. It might be thought that the slowness of the routine is caused by the use of VAL, but in fact this is an insignificant part of the total delay. The reason for this is that, the first time the interpreter encounters line 1000, it carries out the necessary conversion to binary of the counter-values (upper and lower bounds) and step-increment, and stores the results in the variable table (adjusting the amount of memory used to store the values of X, or creating storage, as necessary). Thereafter, each time through the loop, the interpreter refers to these values in the variable table and does not have to use the VAL representation at all.

Actually the routine shown above might be fast enough, but few Adventures are limited to a six-word vocabulary. The real culprit is the FOR/NEXT loop, and the delay will become noticeable as the vocabulary increases to thirty or forty verbs. All the interpreter has to do is add the step-increment to the lower-bound value and compare it to the upper-bound value at the beginning of each loop, but all these calculations are carried out in floating-point arithmetic, and are much slower than operations with integer values. There does not seem to be any escape from this problem and the only general advice is to avoid using FOR/NEXT loops when time is at a premium.

Program development itself can be speeded up by quicker editing, of course, and there are a couple of tips that might be worth mentioning. It may well be that all readers are familiar with these points already, but, at any rate:

(a) You don't have to type in a line to test-run it. That is, if the line:

```
2000  PRINT PEEK L+PEEK M*256
```

already exists in your program, and you want to test that the syntax is correct or that the value you expect to be returned is returned, you can hit EDIT, use down-arrow to bring the cursor down to line 2000, hit EDIT again, and then just delete the first three characters of the line-number. When you hit NEWLINE, the

line is executed in 0/0 mode; the original line 2000 remains in the listing, unaffected.

(b) It follows from the above that line-numbers as well as line texts can be edited. It works like the COPY facility available on some other micros; the original line remains where it was and a duplicate is created with the amended line-number. Sorry to be so obvious.

To conclude the discussion of speeding-up techniques, there are several ways of improving matters that do not incur the increased-space penalty. All subroutines should come at the beginning of a program, preceded by a line such as 1 GOTO 1000.

When the interpreter is searching through the program listing for a particular line number, it always carries out a sequential search from location 16509 onwards; the number of lines it has to check before finding the right one has a noticeable effect on speed. Suppose it has encountered GOSUB 2 and has to pass through line 1 before finding line 2; this will take about 0.022 of a second in SLOW mode. If it has to pass through fifteen lines, the delay increases to about 0.03 of a second. In a two-hundred-line program with multiple GOSUB calls, a considerable improvement will result from putting all subroutines at the beginning.

A similar situation exists with respect to the variable table. Executing PRINT X will cause the interpreter to scan the table from the first byte (the value held at VARS) to the end, looking for the variable called X, and also checking the first few bits of the byte holding X to see whether it has come to the end of the variable name (in the case of variables longer than one letter). Putting the most commonly accessed variables at the start of the table will increase program speed, although not spectacularly. The only way to set up a variable as the first in the table, of course, is to declare that variable before any others; there is no way (without using a machine-code routine) to "undeclare" a variable that has been set up.

Finally, the structure of the variable table should be taken into account when avoiding delays caused by the "shuffling RAM" problem. A great deal of internal housekeeping is performed by the interpreter during certain operations which affect the length of the Display File or the location of the variables in memory, or both. As an example of the first problem, an extra line is added to the File when using SCROLL and the slow-down in execution of a program in SLOW mode is a wondrous thing to behold, making SCROLL almost valueless to the programmer. Regarding the second problem, which has rather more subtle effects, consider what happens when you change the amount of storage that the interpreter needs to hold a given variable in the variable table. Suppose your first variable is X and you give it any numeric value. All goes well until, later in your program, you decide to use it as the control variable in a FOR/NEXT loop. It will now need 18 bytes of storage for the values associated with it (starting value, final value, step increment and the loop-line). This is 12 bytes more than it

needed before, and the whole table has to be moved 12 bytes up in memory to make room for it. Such variables should be located at the end of the table where possible; string variables whose length is liable to fluctuate during program execution should be treated in the same manner.

Programs can only be speeded-up so much in BASIC; after that, machine code must be used for really fast execution. Later in the book, examples of machine-code routines will be given that can be used to speed up sections of a BASIC program. Often, the slowness of a program can be identified and isolated in 10% of the BASIC coding, and it may not be too daunting a task to write a machine-code routine, accessed through USR, to achieve a considerable improvement by replacing the troublesome portion of the BASIC program.

KEEPING THINGS CLEAR

This chapter concludes with an examination of the opportunities offered by ZX81 BASIC for "self-documenting" programming. Ideally, a program listing will contain all the information that a programmer needs to be able to understand how the program works. Although this is an ideal that will seldom be achieved in practice, it is indeed possible to reach a high degree of clarity in a ZX81 program. The designers of the BASIC seem to have tried to include features that would make this possible, perhaps to encourage use of the machine in an educational context.

Unfortunately, the same comments about trade-off that were made earlier in the chapter apply here. To the extent that a program is internally well documented, it will generally run more slowly and take up a great deal more memory. The converse applies as well; the techniques for speeding-up and saving memory that have already been discussed can be used only at the expense of clarity, and a program that has been optimised both for speed and length (so far as this can be done) will be hopelessly incomprehensible without extensive separate documentation. Nevertheless, programmers are in practice bound to use such techniques to achieve their aims and in this sense the design of ZX81 BASIC must be judged to have failed to achieve its objectives.

Apart from the liberal use of REMs in a program, the most valuable aid to clarity is probably the use of long variable names, allowing the programmer to describe the significance of each variable that is being used. Almost as important, however, is the use of variable names for jump and call vectors, which permits the programmer to give a mnemonic description of the function of each subroutine rather in the way that an assembly language programmer can give mnemonic labels to subroutines in an assembler program. Listing 1.2 illustrates the technique with a simple digital-clock program. Whether you find it fully comprehensible is partly to do with how you style your own programs; it makes use of nested GOSUBs to break down the listing into a main routine and a series of subroutines. In a sense this is an unusual application of GOSUB

because each subroutine is only accessed from one place in the program; the idea is that you scan the first half of the program to get an impression of the general flow of logic, then investigate each subroutine to see how the mucky bits of calculation are actually carried out.

Listing 1.2

```
  1 REM DIGITAL TIME-DISPLAY
  2 FAST
  3 REM ************************
  5 REM INITIALISE VALUES
 10 PRINT AT 5,6;"HOUR    MINUTE
  SECOND"
 15 LET A$="   "
 20 LET TIME DISPLAY=200
 30 LET UPDATE=400
 50 LET DISPLAY ROW=10
 55 LET FIRST COL=8
 60 LET MID COL=16
 65 LET THIRD COL=24
 70 LET LOOP=105
 75 LET SECOND=0
 80 LET MINUTE=0
 85 LET HOUR=0
 90 LET NEXT MINUTE=500
 95 LET NEXT HOUR=600
 97 LET NEXT DAY=700
 98 REM ************************
100 SLOW
105 REM LOOP
110 GOSUB TIME DISPLAY
120 IF INKEY$="" THEN GOTO LOOP
130 REM ************************
150 STOP
160 REM **SUBROUTINES********
200 REM TIME DISPLAY
210 PRINT AT DISPLAY ROW,FIRST
COL;HOUR
220 PRINT AT DISPLAY ROW,MID CO
L;MINUTE
230 PRINT AT DISPLAY ROW,THIRD
COL;SECOND
250 GOSUB UPDATE
260 RETURN
300 REM ********************
400 REM UPDATE
440 LET SECOND=SECOND+1
450 IF SECOND=60 THEN GOSUB NEX
TMINUTE
460 RETURN
470 REM ************************
500 REM NEXTMINUTE
510 LET SECOND=0
515 PRINT AT DISPLAY ROW,THIRD
COL;A$
520 LET MINUTE=MINUTE+1
530 IF MINUTE=60 THEN GOSUB NEX
THOUR
540 RETURN
550 REM ********************
600 REM NEXTHOUR
605 LET MINUTE=0
610 LET HOUR=HOUR+1
615 PRINT AT DISPLAY ROW,MID CO
L;A$
```

```
 620 IF HOUR=24 THEN GOSUB NEXT
DAY
 630 RETURN
 650 REM **********************
 700 REM NEXT DAY
 710 LET HOUR=0
 720 PRINT AT DISPLAY ROW,FIRST
COL;A$
 730 RETURN
```

Programs written in this style are of course very laborious to type in, because of the lack of user-definable function keys on the ZX81. Being able to define, for example, shifted-D as equivalent to the string "DISPLAY ROW" would make it much quicker to enter the program and, realistically, users will not use long variable-names unless they are provided with such a facility.

An interesting point arises from observing what happens when you actually run the program; it runs slightly fast for the first minute, then runs slightly slow. The reason for this is the varying amount of time that the interpreter needs to work out what characters to print on the screen when it is asked to print a numeric variable value. In general, it takes the same amount of time to print each character; printing "55" will take roughly twice as long as printing "5". This is bad enough, but the time taken to print "0" is much less than the time taken to print any other number. The explanation is that 0 is stored in a special way, as a sequence of five zeros, rather than according to the normal rules for floating-point variable storage. As soon as the interpreter detects that a number is stored as five zeros, it dumps a "0" to the screen and returns from the floating-point conversion routine in ROM instead of going through the normal sequence of steps in the conversion process.

One can make amendments to allow for this, and indeed Listing 1.1 can be revised to reduce the amount of inaccuracy in timing, but as a general rule one should not attempt to use ZX81 BASIC for accurate timekeeping.

In the next chapter we will be looking at some useful BASIC routines, and employing a couple of additional techniques for improving program clarity. The first is the method of passing parameters; this simply means that a set of variable values is defined just before a GOSUB call to a subroutine which operates on these values. In practice we will often use the method of storing the current variable values in a string. This is partly to reduce memory consumption, but it also makes it much easier to maintain the distinction between local and global variables. That brings us on to the second technique; defining a variable as either local or global and maintaining the distinction throughout a program. Once one begins to write BASIC programs longer than 2 or 3K, the length of the variable table begins to increase alarmingly and it's by no means unusual to end up with a program with a hundred variables. This is all right, but enormous confusion can be caused if a variable is inadvertently used within a subroutine such that its value changes, when the programmer's original intention was to

maintain its value throughout the subroutine call. Even an experienced programmer can easily be tripped up by this particularly time-wasting kind of bug and we'll look at ways of reducing the possibility of its occurrence.

Chapter 2 Subroutines for BASIC programs

There are certain programming tasks which crop up time and time again, and often the best way to handle them is to develop a subroutine which can be used in many different programs. A selection of such subroutines is presented in this chapter. Many of them enhance the capabilities of ZX81 BASIC in the sense that they allow the programmer to achieve more elaborate effects than would otherwise be possible, or "fill in" for statements which are missing from the ZX81's version of BASIC.

INKEY$

From the point of view of user-friendly programming, INPUT has always been one of the weaker aspects of BASIC, and the implementation of it on the ZX81 is particularly poor. The programmer cannot arrange to print text on the screen, immediately before the cursor which prompts for input, to explain to the user the sort of information the program is waiting for. The prompt always appears at the bottom of the screen, eliminating the possibility of dividing the screen display up into windows in which to display various different inputs by the user in a tidy fashion. It is, of course, possible to collect input at the bottom of the screen, so to speak, display it in a reserved area of the screen display, then come back to the bottom of the screen for the next input. Unfortunately, if the user inputs lengthy material, the lower part of the screen display will be wiped out.

The worst feature of INPUT, shared by most implementations of it on popular microcomputers, is that it does not allow the programmer to set restrictions on the kind of data that the user can enter into the program. It is of course possible for the programmer to include, in his program, coding that will examine the user's input and loop back to ask for it to be repeated if necessary. Nothing can be done, however, about the mechanics of the INPUT statement itself, as this is enshrined in a ROM routine. The programmer who employs INPUT effectively hands over control of the program to the ROM during the period in which the user, who may be inexperienced, is trying to enter data. Much depends on the quality of the programming used to implement INPUT on a particular computer. On the ZX81, for example, the line:

```
10    INPUT X$
```

will cause the inverse-L to appear on the bottom line surrounded by inverted commas. The programmer can't choose to have the inverse-L appear without the commas; it is something that always happens when string input is requested. The programmer can choose to ask for numeric input instead, but then the program will crash if the user inadvertently enters a non-numeric character.

The first rule that a programmer learns when writing interactive software is that if a user can get the input wrong, he will. One or two of the programs in this book will feature the INPUT statement, as they are intended to be typed in only as demonstration programs, but it is poor practice to employ INPUT in a program which may be used by a number of other users, unless the implementation of it on a particular computer is unusually good.

INPUT on the ZX81 certainly does not meet this criterion. Woe betide the user who attempts to use the left-arrow key to correct his alphabetic input, and erases the first of the inverted commas; a thorough search through the Sinclair manual will be needed to work out how to escape from the resulting mess.

The solution to the problem is an INKEY$ subroutine, which can be called whenever user input is required. An additional advantage of this procedure is that the programmer can specify a certain number of parameters when he calls the routine, such as the position on the screen where the user's input can be echoed, and the maximum permitted length of input, thus eliminating the possibility that the input will corrupt the screen display. The INKEY$ subroutine is shown in Listing 2.1.

In this listing, parameters are laid out in lines 1-102 for the highest and lowest character codes that the routine will accept as input from the user, the maximum length of input, and the row and column at which the first of the input characters is to appear on the screen. The subroutine itself is in lines 1000-1230 and will build up the user's input in B$, returning automatically once the maximum length is reached, or earlier if NEWLINE is hit. The MAX and MIN values allow only the alphanumeric character set to be input - BASIC keywords and so on are ignored. Other values can of course be chosen, and the programmer may well want to allow punctuation to be input.

Lines 1100-1110 allow the user to cancel his input by using shifted-left arrow as a backspace; the last character input is subtracted from B$, the screen display is adjusted, and a test is made that B$ actually contains at least one character that can be subtracted.

The subroutine therefore allows the programmer to designate screen areas as windows in which specified user input will appear, without disturbing the contents of the rest of the screen. There is, however, no specific check that the end of a screen line has been reached - the routine will not work properly if it is asked to print beyond the final column on the screen, so this should be taken into account when planning the screen format.

Listing 2.1

```
  1 LET MAX=63
  2 LET MIN=28
  4 LET LEN=20
100 LET R=4
102 LET C=4
110 GOSUB 1000
112 PRINT AT R,C;" ";
120 PRINT B$
130 STOP
1000 LET B$=""
1002 LET Z$="■"
1005 SLOW
1010 PRINT AT R,C;
1012 IF LEN B$=LEN THEN RETURN
1015 GOSUB 1400
1020 IF CODE A$=118 THEN RETURN
1040 IF CODE A$=114 AND B$>"" TH
EN GOTO 1100
1050 IF (CODE A$>MIN) AND (CODE
A$<MAX) THEN GOTO 1200
1060 GOTO 1010
1100 LET B$=B$( TO (LEN B$-1))
1101 PRINT AT R,C;" ";
1102 LET C=C-1
1110 GOTO 1010
1200 LET B$=B$+A$
1210 PRINT AT R,C;A$;
1220 LET C=C+1
1230 GOTO 1010
1400 LET Z$=("■" AND Z$=" ")+("
" AND Z$="■")
1420 PRINT AT R,C;Z$;
1425 LET A$=INKEY$
1427 IF A$="" THEN GOTO 1400
1430 RETURN
```

Since user input may now appear anywhere on the screen, it's really essential to give the user a cursor which directs his attention to the right area. The nested subroutine at 1400-1430 does this; a black square is used but any other symbol can be selected. Owners of more expensive machines than the ZX81 are accustomed to luxuries like flashing cursors, but there is no reason why the ZX81 owner should do without such a facility, nor is machine-code needed to provide it. Line 1400 has the effect of flashing the block on and off while the subroutine is active.

Normally the programmer will call the INKEY$ subroutine from a number of areas within the main program, with new values for the parameters each time. To save space, however, the parameters can be compressed into one line and stored in a string, as mentioned at the end of the previous chapter. Lines 1-102 in Listing 2.1 can then be replaced by the single line:

```
1       LET P$="6328200404"
```

In order to extract the various parameters, the TO slicer can

be used. Here are a couple of the appropriate amendments to Listing 2.1:

```
112    PRINT AT VAL P$(7 TO 8),VAL P$(9 TO );" ";
...
1012   IF LEN B$=VAL P$(5 TO 6) THEN RETURN
```

Obviously this makes the subroutine a bit clumsier, but after all it only occurs once in the program whereas the parameters may have to be specified a dozen times or more. There is, however, a more important point; only one (string) variable is being used to store the parameters instead of five numeric ones. It's much less likely that the programmer will inadvertently use this single variable within the subroutine, or within some nested subroutine that is called by the first; all he has to do is remember to avoid using P$ again.

There are one or two dialects of BASIC that allow the distinction between local and global variables to be maintained, but in the case of ZX81 BASIC it's the programmer's responsibility to keep clear on the difference in order to avoid the sort of confusion that was mentioned at the end of Chapter 1. If the technique described above is followed throughout the program, a relatively small number of string variables will be used which can be redefined anywhere within the main program; a separate set of variables (both string and numeric) will be used only within subroutines. Provided the distinction is maintained by the programmer throughout, a great deal of debugging effort can be avoided.

PRINT USING

This is one of those useful commands that is usually found only in 12K or "business" BASICs. It allows the programmer to predefine a screen format which will govern the display of either numeric or string values, as they may be generated in the course of the program. There are various different forms of PRINT USING in various dialects, and some of the more esoteric versions may be thought to be more trouble than they're worth. It is, however, very useful to be able to predefine the number of decimal digits that will appear on the screen on either side of a decimal point, and this can be done quite easily with a BASIC subroutine; it's also worthwhile to devise ways of picking out defined numbers of characters from a string, especially when fitting text into predefined windows on the screen.

Listing 2.2 accomplishes the first task, and allows the programmer to define any number of digits to appear both on the left and on the right of a decimal point when a numeric value is printed on the screen. It's surprisingly long, due mainly to the need to cope with all sorts of input including integers and values less than 1. It's also, frankly, something of a spaghetti program, but it's difficult to see how this can be avoided; any reader who believes he can tidy it up is invited to do so. At any rate, it works; we will give a brief explanation of it.

Listing 2.2

```
   10 LET A=65.333
   20 LET P$="▒▒▒▒.▒▒"
   50 GOSUB 1000
   70 STOP
 1000 REM
 1100 LET A$=STR$ A
 1110 LET L1=LEN A$
 1120 LET L2=LEN P$
 1150 LET X=0
 1160 LET Y=0
 1170 FOR R=1 TO L1
 1180 IF A$(R)="." THEN LET X=R
 1190 NEXT R
 1200 FOR R=1 TO L2
 1210 IF P$(R)="." THEN LET Y=R
 1220 NEXT R
 1300 IF X=Y THEN GOTO 1400
 1310 IF (X>Y) OR ((NOT X) AND L1
>Y-1) THEN GOTO 1390
 1315 IF NOT X THEN GOTO 1342
 1320 FOR R=1 TO Y-X
 1330 PRINT " ";
 1340 NEXT R
 1342 FOR R=1 TO Y-L1-1
 1344 PRINT " ";
 1346 NEXT R
 1350 GOTO 1400
 1390 PRINT "*";
 1400 IF NOT X THEN PRINT A$;
 1410 IF X THEN PRINT A$(1 TO X);
 1420 IF NOT X THEN GOTO 1482
 1450 IF (L1-X)=(L2-Y) THEN GOTO
 1600
 1460 IF L1-X>L2-Y THEN GOTO 1700
 1470 LET R=(L2-Y)-(L1-X)
 1480 PRINT A$(X+1 TO L1);
 1481 GOTO 1490
 1482 PRINT ".";
 1484 LET R=L2-Y
 1490 FOR S=1 TO R
 1500 PRINT "0";
 1510 NEXT S
 1520 RETURN
 1600 PRINT A$(X+1 TO L1);
 1610 RETURN
 1700 FOR R=1 TO (L2-Y)
 1702 PRINT A$(X+R);
 1705 NEXT R
 1710 RETURN
```

The subroutine, like any subroutine for simulating PRINT USING in relation to numeric values, starts by using STR$ to get the string equivalent of the value. Now it can divide it into a number of characters before and after the decimal point. X is the number of characters before the point, and Y is used to hold the number of digits that will be displayed before the decimal point when the number is printed out. The specification of the format of the printed value is held in P$ and so this is an argument that is passed to the subroutine by the programmer; P$ is a string of black squares containing a decimal point, although as a matter of fact the routine only looks at the length of the string either side of the decimal point and any symbol would do in place of black squares.

Lines 1150-1220 build up values for X and Y, and the routine then follows different paths depending on the nature of the original numeric value. The condition NOT X, for example, is satisfied when X=0, that is, there are no digits to the right of the decimal point.

The routine is quite flexible and will accept any number of characters either side of the decimal point in P$, including zero characters. The conventions it uses for dealing with various types of value have been arranged to resemble the Microsoft BASIC version of PRINT USING and are as follows:

If the numeric field (the number of characters) to the left of the point is greater than the number of digits in the original numeric value, the unused field positions to the left of the number will be filled with spaces. It's this feature which allows one to use the routine to align a column of values so that the decimal point (which may of course represent a division between pounds and pence) is in the same position on each line.

If the numeric field to the right of the point is greater than the number of digits, the unused field positions are filled with zeros. Again, the value "17" will come out as, for example, "17.00" which is useful if it represents an amount in pounds and pence.

Field overflow occurs when the number of digits to be displayed is greater than the field in P$ that has been allowed for them. If overflow occurs in the digits before the point, a warning asterisk is printed followed by the whole number. If the overflow is to the right, the surplus decimal places are simply not shown. Again, this is useful for displaying sterling amounts. The only weakness of the routine is that it does not perform rounding-up or rounding-down before truncating the digits in this way. However, any necessary adjustment can be carried out to the value before the subroutine is called; Chapter 3 will cover this point.

The other routine that is worth implementing is one to select characters from a string to fit into predefined fields. Listing 2.3 shows a subroutine to do this, in which line 10 sets up the original string in A$ and line 20 defines the field as an argument to be passed to the subroutine. There is no significance in the symbols used in P$ to define the number of characters which will be printed from each word in A$; any symbol can be used in place of the black squares. The full stops are, however, significant as line 1405 of the subroutine looks for these, and then line 1830 prints them after the words of the original string.

The uses of the subroutine should become clearer if it is typed in and tested with various different versions of P$. In the example shown, the output is:

```
JO.PA.JONES
```

It doesn't matter how much longer the words in A$ are than the number of black squares in P$; the routine will always truncate

them to the specified length, and then print the full stop (obviously some other symbol than a full stop can be used if lines 1405 and 1830 are altered). The routine works by building up a number of pointers; P1 and P2 point to the start and end of each word in A$, and PP1 and PP2 are used similarly for P$. PPT holds the length of each set of black squares, and PT is incremented to show the number of characters of each word that have so far been dumped to the screen; line 1560 tests one value against the other to see if the maximum number of characters has been printed yet.

Listing 2.3

```
   10 LET A$="JOHN PAUL JONES"
   20 LET P$="■■.■■.■■■■■■■■■"
   50 GOSUB 1000
  100 STOP
 1000 LET P1=1
 1010 LET P2=1
 1020 LET PP1=0
 1030 LET PP2=1
 1040 LET 0=0
 1050 GOSUB 1200
 1060 IF 0=1 THEN RETURN
 1070 GOTO 1050
 1200 GOSUB 1300
 1210 GOSUB 1400
 1215 LET PPT=PP2-PP1
 1220 GOSUB 1500
 1230 LET P1=P2
 1240 LET PP1=PP2
 1250 RETURN
 1300 IF P2>LEN A$ THEN GOTO 1600
 1305 IF A$(P2)=" " THEN GOTO 135
0
 1310 LET P2=P2+1
 1320 GOTO 1300
 1350 LET P2=P2+1
 1360 RETURN
 1400 IF PP2=LEN P$ THEN GOTO 160
0
 1405 IF P$(PP2+1)="." THEN RETUR
N
 1410 LET PP2=PP2+1
 1420 GOTO 1400
 1500 LET PT=1
 1510 PRINT A$(P1);
 1520 LET PT=PT+1
 1530 LET P1=P1+1
 1540 IF P1>LEN A$ THEN GOTO 1600
 1550 IF P1>P2 THEN RETURN
 1560 IF PT>PPT THEN GOTO 1800
 1570 GOTO 1510
 1600 LET 0=1
 1610 RETURN
 1800 IF PP2+1>LEN P$ THEN GOTO 1
600
 1830 PRINT P$(PP2+1);
 1840 LET PP2=PP2+1
 1850 RETURN
```

Since there are nine black squares in the final field of P$, the routine will show any surname up to that length if indeed it is to be used to display proper names; there are numerous applica-

tions for a routine of this sort and the programmer may well amend it to suit his purposes. For example, line 1830 can be altered so that a space is printed after each full stop and before the beginning of the next character.

DATA STATEMENTS

Finally, a routine to implement DATA on the ZX81 would be of particular value to almost any programmer; DATA, READ and RESTORE were omitted from the 8K BASIC owing to shortage of space. Various methods have been put forward to provide this facility and these will be examined; in the end, though, a machine-code routine is the only satisfactory way of doing it, and information will be included in the later chapters of the book on the techniques needed to write such a routine.

ZX81 programs are sometimes encountered in which the programmer has chosen to POKE byte values into either a string or a REM line, so that they can be extracted later by PEEK. Usually, when this is done, the information is stored in the first line of the program; this is because the RAM address of the first byte of the data can easily be calculated. If a REM statement is used, the first byte following the REM token will always be at memory location 16514, and the rest of the program can be edited at will without disturbing this situation.

This is, however, not the best way to store data of the kind normally found in DATA statements on other computers. The point of DATA statements is that they provide easily accessible storage for the data that a program uses, in the sense that the programmer only has to LIST the program to check on the values currently being used as a database by the program. This becomes much more difficult if the programmer is confronted by a REM line consisting mainly of graphics blocks, BASIC reserved words and question marks; that is, of course, what happens when numeric values in the range 0-255 are stored in individual memory locations and then printed to the screen by the LIST command. A further problem is that editing such a line becomes unbearably tedious as soon as the amount of data to be stored exceeds a few dozen bytes, due to the primitive editing facilities on the ZX81. Large quantities of such data should be split up by being stored in separate lines, but of course it is then more awkward to calculate the memory location of the first byte of each chunk of data within the program listing.

A related problem is that the data must be regarded as one big block. In many programs there may well be a database consisting of, say, five chunks of data which all serve different functions, and the programmer would like to access, for example, the fourth such chunk independently of the rest of the database. As a matter of fact this is a weakness in the original version of Microsoft BASIC's implementation of DATA; the program always has to restore the data pointer to the beginning of the listing, then search through the DATA lines looking for some key-identifier which signifies the beginning of the chunk it needs to use.

The one advantage of the method of storing data by POKEs is that it saves space, but this is a benefit that generally has to be given up when one starts using DATA statements in any dialect of BASIC; they are notoriously space-consuming and this characteristic is inseparable from the features of ease of access and editing mentioned above.

Listing 2.4 shows one way of storing data in a string, a fairly flexible method in that the data items can be either numeric or alphabetic and in either case can be made up of any number of characters. Having set up the string in line 20, the variable P is set to zero (meaning that we want to search right through the string from the beginning) and the subroutine at line 1000 is called. The listing is a demonstration program showing how the subroutine can read through data items, both numeric and alphabetic, looking for the item "MARKER" which marks the position in the data list of the item we are after; the main program then calls the subroutine one more time and returns with the value 66 in the variable V. Note the conventions that must be followed; numeric data is followed by a comma, and alphabetic data by a semi-colon. Note also that there is no "out-of-data error" message if an attempt is made to read one more item than the number of items stored in the string.

Listing 2.4

```
  20 LET Z$="12,STRING;45,877,MA
RKER;66,3,"
  30 LET P=0
 100 GOSUB 1000
 110 GOTO 100+(20*(V$="MARKER"))
 120 GOSUB 1000
 170 PRINT "DATA ITEM IS:";V
 200 STOP
 999 REM ******************
1000 LET V$=""
1020 LET P=P+1
1030 IF Z$(P)="," OR Z$(P)=";" T
HEN GOTO 1100
1040 LET V$=V$+Z$(P)
1050 GOTO 1020
1100 IF Z$(P)="," THEN LET V=VAL
 V$
1120 RETURN
```

Z$ can be as long as we like and we don't have to set P to zero each time we examine it; we can carry on from the value of P that was reached on the last occasion that the subroutine was used, and avoid having to read data that was previously read. In practice one would not want to make Z$ too long because of the tedium involved in editing the line, referred to above. Instead, a number of string variables can be used and each one can be equated with Z$ in turn, just before the routine is called.

There are not many string variable names available in ZX81 BASIC and it may be necessary to use a string array if there is a large quantity of data. The procedure is much the same. In both

cases a lot of memory is used up because the text of each string duplicates itself in the variable table as soon as the program is run. As a matter of fact there is a compensating advantage in this; it would not be too difficult to write a set of subroutines in BASIC to allow one to edit the contents of string variables. Typical commands that one might want to implement would be Search, Delete and Change, which are available within the editing facilities of other computers. Normally one summons up the command with a single keystroke, giving the initial letter of the command, and arguments can be added; for example, Sw will search for the first occurrence of the letter W in the string, moving the cursor to that location in the string. A BASIC program to implement these facilities would print the contents of the string on the screen and provide a cursor under keyboard control which would be moved back and forth within the string by the user. The catch is, of course, that the new version of the string would not exist in the program listing although it would be easy enough to enter it into the variable table. What sort of task would be involved in writing a BASIC program to find the BASIC line-number containing the original string so that it could be overwritten with the new version, moving the subsequent lines up or down in memory to allow for the different length of the new line? It's essentially quite a straightforward programming exercise provided that the manner in which BASIC programs are stored in memory is clearly understood. Later in this book some information will be given on the technique of writing self-modifying BASIC code of this kind.

For the moment, however, I would like to illustrate a different method of storing DATA within BASIC program listings which does not seem to have been put forward previously, and yet which offers some advantages over the methods so far discussed. The essential idea of the technique can be seen from the short routine given as Listing 2.5. In this routine, a memory address stored in locations

Listing 2.5

```
  10 SLOW
  20 LET A$="11+PEEK 16402+PEEK
16403*256"
 100 LET P=VAL A$
 110 REM ▄▀▄▀▄
 200 GOSUB 900
 400 LET Q=VAL A$
 500 REM ▓▓▓▓▓▓
 550 LET P=Q
 600 GOSUB 900
 710 STOP
 899 REM ****************
 900 PRINT
 905 FOR X=0 TO 3
 910 PRINT CHR$ PEEK (P+X);
 920 NEXT X
 930 RETURN
```

16402/3 is used; this is the address of the numeric or string variable currently being interpreted as the BASIC program is executed, line by line. Notice how we make use of the ZX81's expression evaluation capability by storing the formula which derives this address in A$ as a string of BASIC keywords and

associated data. We can then call up the value currently stored in 16402/3 by using VAL A$. In line 100, the address that is produced is the memory address within the BASIC listing of the character P which represents the variable-name; if we had used a multi-letter variable, by the way, the address would have been that of the first letter. In the REM line which follows immediately, the first of the graphics characters is 11 bytes further on in the listing from the character P; hence, the value produced by the complete formula is the memory address of the start of this sequence of graphics characters. In the listing, which is only a short demonstration program, the sequence of graphics characters is printed out on the screen by the subroutine at 900.

You may be wondering why a different variable (Q) is used on the second occasion that the technique is used in the listing; it means that we need an extra line 550, transferring Q back to P, in order to be able to use the same subroutine again. The answer is that the functioning of the ZX81 interpreter compels this. The first time that any variable name is encountered by a BASIC program, the address stored in 16402/3 is the address of the name in the program listing. However, if the same variable-name occurs later on in the same program, when it has already been set up in the variable table, then the address that is stored is that of the variable's location *within the variable table*. In other words, if we want to be able to use this technique to make the program itself calculate where we have stored data within REM lines, we must use a new variable name each time.

If we intend to use the technique to store data in the manner of a DATA statement, some decisions have to be made about the method of storage. The most economical method would be to store the values by POKEing them into the REM lines. However, we won't then be able to read what is stored easily by LISTing the program, and calculations will have to be made by the program to find the value of any item of data which is stored in two bytes or more because its value exceeds 255. Storing alphabetic data presents less of a problem. Storing machine-code routines is another obvious application; each routine can be located in a REM line at the position in the BASIC program where it is actually intended to be used, and executed with a line such as:

```
120    RAND USR P
```

which is a much neater approach than storing all the routines at the start of the program and having to work out where each one begins in memory.

The advantages of the technique are that it avoids duplicating the data since the material in the REM lines doesn't exist anywhere else in memory, and that it makes the program easier to follow because each chunk of data can be located in that section of the program that uses it. It is like having a much more flexible RESTORE-pointer than that available in the conventional DATA/READ/RESTORE implementation of BASIC; the address returned by VAL A$ will be a pointer to the beginning of the chunk of data occurring

in the REM line that follows it and there is no need to read right through all the data in the program to get there. The disadvantage becomes clear when you want to read the same chunk of data twice; the only way to do this is to insert a CLEAR statement in the program to make BASIC forget about the variable-name that it has previously encountered. It may be preferable to use the alternative way of finding the address of a line within a program listing, described in Chapter 1: use the values in locations 16425/6 to calculate the address of each REM line, then PEEK at the values stored in that line. Whichever approach is adopted, the programmer cannot expect lightning-fast execution of his READ facility so long as it is carried out entirely by BASIC coding.

As a final point, consider the rather odd little program in Listing 2.6. It prints a "5" on the screen, according to the

Listing 2.6

```
  20 LET A$="1+PEEK 16402+PEEK 1
6403*256"
 100 LET P5=VAL A$
 200 LET K=PEEK P5
 300 PRINT CHR$ K
```

following logic: the value calculated by A$ is the address of the byte following the P in line 100. At this address is the second character of this two-letter variable; the rest of the program PEEKs it and then prints its CHR$ representation which is, of course, 5. The point of all this is that at no time is the six-byte floating-point representation of "5" contained in the BASIC listing; this is a way of slipping numeric values into ZX81 programs without paying the six-byte cost. Variable-names can be any length so, in theory, you could store a couple of thousand bytes of data in an enormous variable-name and use a routine similar to Listing 2.6 to extract them. This is not necessarily recommended - it would certainly produce an odd-looking program - but it's an idea to bear in mind.

Chapter 3 Business applications

The first comment to be made about use of the ZX81 as a business tool is that its application in this way must be strictly limited; there is no point in over-optimism about its power. Apart from slow execution speed and limited memory, problems are likely to be encountered with any program that needs to hold values representing amounts of money, because the five-byte floating-point arithmetic format is the only one available on the machine, and this can, of course, produce errors when one is attempting to calculate exact amounts.

The previous chapter mentioned the problems that can arise with rounding-up and rounding-down operations. If an amount in pounds and pence is represented as a numeric variable with numbers on both sides of the decimal point, and is then multiplied or divided, the result will often be a value with a large number of decimal places. If INT is used to correct this value, the value is effectively truncated, all numbers to the right of the decimal point being lost.

The conventional advice in this case is to handle amounts representing money by multiplying all values by 100 to start with, then dividing the result by 100 at the end. In this way the program is handling integers which represent the number of pennies in the various amounts. Divisions will still produce results with many decimal places and INT is used to remove these, after which division by 100 can be performed to show the result as pounds and pence, but in order to cope with fractions of a penny, the value .5 is added to the result before INT is applied to it. Listing 3.1 shows the procedure, which should be reliable in all cases; the result produced by this method can be passed to a subroutine like the PRINT USING routine shown in Chapter 2, so that it can appear neatly on the screen within a column of figures.

Listing 3.1

```
10 LET AMOUNT=1.375
20 LET AA=AMOUNT*100
30 LET DIV=3
40 LET RESULT=AA/DIV
45 LET ROUND=INT (RESULT+.5)
50 LET ANSWER=ROUND/100
60 PRINT ANSWER
```

These problems, by the way, are inherent in floating-point representation of values and have nothing to do with differences between the old ROM version of the ZX81, which had an error in the arithmetic routines, and the new ROM version.

DEF FN

Anyone using a ZX81 either for business or scientific applications will probably notice the lack of this command in ZX81 BASIC more than any other. In a way this is strange, because DEF FN does nothing that cannot be done easily enough by other means in any dialect of BASIC. To clarify matters for users who have not come across DEF FN before, it allows you to avoid quoting the same formula over and over again within a BASIC program whenever you want the formula to be evaluated by the program with the current values of the variables cited in the formula. A simple example would be:

```
10      DEF FNA=B*C/SIN 4.896
...
100     FOR B=1 TO 10
110     FOR C=1 TO 10
120     PRINT FNA;" ";
130     NEXT C
140     NEXT B
```

The same effect could be achieved simply enough by replacing the FN statement in line 120 with the formula itself. The advantages of DEF FN only become apparent when the formulas become extremely long and complicated, as in many scientific programs, and when they have to be repeated several times within the program; instead of typing in the whole formula again, the programmer just "refers" the interpreter back to the line of the program in which the formula was quoted, and in effect instructs it to work it out again using the current values of the variables mentioned in it.

A somewhat more sophisticated version of DEF FN, found in DEC BASIC, allows the programmer to forget about the precise variable names he will be using in his program at the time that he types in his DEF FN formulas, usually at the beginning of the program. All he has to decide is the number of variables that will be quoted in the formula. In the example already given, we would say:

```
10      DEF FNA(N1,N2)=N1*N2/SIN 4.896
```

and then in line 120 he would code:

```
120     PRINT FNA(B,C);" ";
```

The advantage of this is that the same formula can be used elsewhere in the program, even if B and C only occur in one part of the program. The variables shown in brackets in line 120 are used by the program to replace the "dummy" values N1 and N2 in the formula, and if two entirely different variables were shown in the brackets, those would be used instead.

Even this version of DEF FN can be replaced by a program structure that sends control to a subroutine which contains the formula to be worked out. It may, however, be necessary to transfer values from the variables currently in use by the main program, into the variables that are quoted in the subroutine, and then collect them back again when return is made from the subroutine.

However, there is a method of implementing DEF FN in ZX81 BASIC that is somewhat more elegant than the techniques described above. Once more, it relies on expression evaluation. Listing 3.2 illustrates how it's done. The formula is stored as a string in an array, and in this example we have allowed for ten such formulas, each with a maximum length of 15 characters or, strictly speaking, 15 bytes of storage (a mixture of one-byte keywords and numerical data).

Listing 3.2

```
   1 LLIST
   5 DIM Z$(10,15)
  10 LET Z$(1)="B*C/SIN 4.896"
  15 LET Z$(2)="B**3+(C>B AND B=
3)"
  20 LET B=3
  30 LET C=4
  40 LPRINT VAL Z$(2)
  50 LPRINT VAL Z$(1)
 100 LET B=3.5
 110 LPRINT VAL Z$(2)
217
-12.205159
216
```

We can then change the values of the variables used by the program, and whenever we call up the numerical value represented by the evaluation of the expression stored in one of the array elements, the program will use the current variable values. In the example, the second part of the expression stored in Z$(2) evaluates as 1 (logical TRUE) when B=3, giving a total of 217; changing B gives a logical FALSE with a numeric value of zero.

The simpler form of DEF FN, described at the beginning of the chapter, can therefore be simulated with statements of the form "LET X=VAL Z$(nn)", where nn is the array element containing the desired formula. The statements setting up the formulas can be put at the beginning of the program, just after the DIM statement, which will help clarify certain types of program. It isn't possible, however, to simulate the DEC BASIC form of DEF FN.

The method appears wasteful of memory because the formulas are present both in the program listing and in a string array in the

variable table, but don't forget that we avoid losing six bytes of memory for storage of each numeric value by using this technique. How much it slows the program down will depend on the kind of expressions used; for example, if the formula includes exponentiation, this is already so slow that the user probably won't notice the extra delay while the ZX81 converts the string expression into its numeric equivalent.

STORING DATA ON TAPE

Any business program is likely to rely on some kind of database which needs to be stored independently of the program that manipulates it. ZX81 users can, however, only store data in one way: put it in the program, in REM statements or perhaps long strings, or into the variable table; then save the whole BASIC program to tape each time. I'm afraid that there are no clever techniques to get around this problem; however, only a short machine-code routine is needed to make a copy on tape of any selected area of memory, and the necessary ROM calls will be discussed in Chapter 7.

PRINTER FORMATTING

The addition of the ZX printer to the Sinclair range facilitates a number of typical business computer applications. Program output can be dumped to the printer routinely with LPRINT; graphical displays such as bar charts can also be hardcopied. On many computers, a special routine needs to be worked out to transfer a screen display to the printer, but the COPY command gives the user the ability to do this with a single program line.

Nevertheless, the ZX printer is a severely limited device and the most appropriate type of application for it, apart from documenting programs, is typified by the mailing list program, which provides a useful business tool without straining the machine's abilities. Various listings have been published which illustrate this kind of program, but let's focus on two parts of such a program which sometimes give trouble; formatting the output of a list of names and addresses, and structuring the program so that it can easily be understood and amended.

Listing 3.3 shows an LPRINT routine for producing names and addresses in a neat format from a mailing list, together with a test string with some "awkward" elements in it, such as a lengthy hyphenated name. Each name and address can either be stored as a string literal or input to an array, but in either event the amount of typing to be done has been reduced to a minimum; a carriage return is signified by a "/" and the routine will take care of the need to spread some lines of the string over two lines of the printout. The routine also spaces out the consecutive lines of the printout to make the result as clear as possible.

Note the method of making the printer print several blank lines - this is done by feeding it with a series of commas, where each

pair of commas forces a single blank line. In line 1086, enough carriage returns are forced to make the single name-and-address feed up clear of the paper-tear bar, but if printing a whole sequence of names-and-addresses, you would probably want to reduce the number of carriage returns somewhat.

Listing 3.3

```
   20 LET A$="LIEUTENANT-GENERAL
A. B. ALVAREZ-MENDOZA/23 THE ROO
KERIES/NEWCASTLE-ON-TYNE NE1 3XX
/ENGLAND"
  100 GOSUB 1000
  110 STOP
 1000 LET E$=""
 1010 FOR A=1 TO LEN A$
 1022 GOTO 1080+(10*(A$(A)="/"))
 1080 LET E$=E$+A$(A)
 1082 NEXT A
 1084 GOSUB 1100
 1086 LPRINT ,,,,,,,,
 1088 RETURN
 1090 GOSUB 1100
 1094 GOTO 1082
 1100 LPRINT
 1105 IF LEN E$<33 THEN GOTO 1200
 1110 IF E$(32)=" " THEN GOTO 130
0
 1120 FOR Q=32 TO 1 STEP -1
 1130 IF E$(Q)=" " THEN GOTO 1400
 1140 NEXT Q
 1150 GOTO 1500
 1200 LPRINT ,,E$
 1205 LET E$=""
 1210 RETURN
 1300 LPRINT ,,E$( TO 31)
 1320 LPRINT ,,E$(32 TO )
 1335 LET E$=""
 1340 RETURN
 1400 LPRINT ,,E$( TO Q)
 1420 LPRINT ,,E$(Q+1 TO )
 1435 LET E$=""
 1440 RETURN
 1500 SLOW
 1510 PRINT "STRING TOO LONG"
 1520 STOP
```

```
LIEUTENANT-GENERAL A. B.
ALVAREZ-MENDOZA

23 THE ROOKERIES

NEWCASTLE-ON-TYNE NE1 3XX

ENGLAND
```

The only eventuality that the routine can't cope with properly is a sequence of more than 32 characters with no spaces. A further subroutine could be added to it to deal with this, but it really is an unlikely eventuality.

The routine works by breaking down the problem into successive steps. We only have to worry about one set of characters between oblique-strokes at a time, so lines 1000-1082 will chop out each such set, and line 1100 is called as a subroutine. If there are only 32 characters to think about then the routine at 1200 can just print them, otherwise we have to spread them over two lines. If the first line happens to coincide with the end of a word then the routine at 1300 can cope, otherwise we have to count back along the string to find a space, which signifies the end of a word, and then the routine at 1400 will deal with printing the two lines in such a way as to avoid coming to the end of a line in the middle of a word. In each case return is made directly from the routine to the line that called it with GOSUB. Finally, line 1500 will handle the notional case of a string more than 32 characters long without spaces, which is more likely to be a mistake by the user than a valid name or address.

NESTED MENUS

A program such as a mailing list utility is likely to need amendment from time to time as the nature of the business changes; the method of program structuring suggested here relies on a set of menu-structures to guide user choice, nested as necessary whenever the range of choices becomes too great. By "too great" we really mean that the program listing itself would become too complex if we tried to arrange it in such a way that the user could, by hitting a single key, select one from a wide range of possible options. Instead, we make the user step through a sequence of option-choices each time he wants to use the program, thus narrowing down the possibilities to the one he wants. Although this makes the user do a little more work (hit a few more keys), it's likely to save time in the long run. The well-known acronym KISS (keep it simple, stupid) applies here; apart from keeping the program simple, we want to make it unlikely that the user himself will make errors in option-choice. Otherwise he will sometimes have to retrace his steps to correct errors and in addition we would have to add a great many lines to the program, checking for erroneous choices which might, under certain circumstances, cause the program to go wrong; and on top of that, we would have to add a set of error messages telling the user why his mistaken choice wasn't being obeyed and what he was doing wrong. By the time all that had been fed into the program listing, there wouldn't be much room left for data.

As an example of a nested-menu structure, Listing 3.4 shows the beginning of a mailing-list input, update and printout program, and the screen display that it produces. Note how the half-line commas are used in lines 110 and 120 to space out the display without any need for separate LPRINT lines, and without having to use PRINT AT. TAB is generally a quicker and less space-consuming way of constructing ordered rows and columns of data on the screen than PRINT AT.

Listing 3.4

```
  100 CLS
  102 LET T=29
  108 SLOW
  110 PRINT ,,TAB 5;"MAILING LIST
  PROGRAM",,,,TAB 0;"ENTER NEW FI
LES";TAB T;"1",,TAB 0;"AMEND CUR
RENT FILES";TAB T;"2",,
  120 PRINT TAB 0;"SAVE FILES TO
TAPE";TAB T;"3",,TAB 0;"LOAD FIL
ES FROM TAPE";TAB T;"4",,TAB 0;"
HARDCOPY FILES";TAB T;"5",,,
  130 PRINT ,,"PLEASE TYPE A NUMB
ER FROM 1 TO 5",,"CHOICE:   ";
  135 LET C=CODE INKEY$
  140 GOTO 135+10*(INKEY$<>"" AND
  C>26 AND C<34)
  145 PRINT CHR$ C
  150 GOTO 1000*(C-28)
 1000 REM OPTION 1
 1010 STOP
 2000 REM OPTION 2
 2005 COPY
 2010 STOP
 3000 REM OPTION 3
 3010 STOP
 4000 REM OPTION 4
 4010 STOP
 5000 REM OPTION 5
```

```
      MAILING LIST PROGRAM

ENTER NEW FILES                  1

AMEND CURRENT FILES              2

SAVE FILES TO TAPE               3

LOAD FILES FROM TAPE             4

HARDCOPY FILES                   5

PLEASE TYPE A NUMBER FROM 1 TO 5

CHOICE:   2
```

Lines 135 to 150 accomplish several things: they wait until the user hits a key, then check that it's in the range that the program expects - otherwise it just cycles back to 135 again. The input is printed on the screen, and then the program branches to one of five subroutines depending on the value chosen. However, the program is structured to jump from one routine to another using GOTO rather than GOSUB. Although this may seem less lucid, it means that we will later be able to jump back from any subroutine to the main menu without having to worry about one too many or one too few RETURNs. To add clarity to the program we can, of course, predefine the line numbers we're jumping to as variable names, but then we can't use the branching-technique of line 150. If you prefer to structure the program with nested GOSUBs, this only requires minor amendments to the structure suggested here.

Nesting is going to come into it because we have not outlined the full range of choices available to the user in this first screen display. The option "AMEND CURRENT FILES" breaks down into several sub-choices, and if the user takes this option he will be presented with a sub-menu as in Listing 3.5 (after a suitable delay to show him the choice-number that he has picked, on the bottom line of the first screen display).

Listing 3.5

```
  100 REM MAIN MENU
  110 STOP
  960 LET T=29
  990 SLOW
 2000 CLS
 2010 PRINT ''TAB 10;"SUB-MENU TW
O";''TAB 0;"DELETE ONE FILE";TA
B T;"1";''TAB 0;"ADD TO EXISTING
FILES";TAB T;"2"
 2020 PRINT TAB 0;"CHANGE ONE FIL
E";TAB T;"3";''TAB 0;"RETURN TO M
AIN MENU";TAB T;"4"
 2030 PRINT '';"PLEASE TYPE A NUMB
ER FROM 1 TO 4";''"CHOICE:  ";
 2040 LET C=CODE INKEY$
 2050 GOTO 2040+20*(INKEY$<>"" AN
D C>28 AND C<33)
 2060 PRINT CHR$ C
 2070 GOTO 2100+100*(C-28)
 2200 REM OPTION 1
 2210 GOTO 100
 2300 REM OPTION 2
 2310 STOP
 2400 REM OPTION 3
 2410 COPY
 2420 STOP
 2500 REM OPTION 4
 GOTO 980
```

```
          SUB-MENU TWO

DELETE ONE FILE                1

ADD TO EXISTING FILES          2

CHANGE ONE FILE                3

RETURN TO MAIN MENU            4

PLEASE TYPE A NUMBER FROM 1 TO 4

CHOICE:  3
```

The sub-menu identifies itself as such, with a number (TWO) so that further sub-menus can quote sub-menu two as an optional choice for the user to jump to. It's also important to give the option to return to the main menu; the user can always avoid confusion if he works his way back to the main menu and then starts following his path of option-choices again (and much the same comment applies to the programmer).

Programs written in this way have a nice modular structure such that one can tack extra bits onto them at will as the need arises; if the distinction between the main menu and the sub-menus is maintained, no problems should arise. If necessary, sub-sub-menus can be created, although there is a limit to the complexity of program that the ZX81 will be able to hold in memory.

WINDOWS

In the examples given so far, the user has not needed to hit more than one key to indicate what he wants to do. Inputting data is a different matter and some care must be taken to display the data in the right place on the screen. We have already given an INKEY$ routine to collect multi-character input, but if the input breaks down into different types of data, then screen "windows" must be worked out by the programmer to arrange the data in such a way that the user has the best chance of spotting errors. Listing 3.6 shows a typical data entry routine in a program intended to store employee information. The INKEY$ routine is called repeatedly, in an amended version which allows different parameters to be passed to it for each item of data. Two of these parameters are row and column values for PRINT AT, and they ensure that the data appears in the appropriate screen location. Another parameter, length, is set so that the data display cannot overlap the text printed on the screen by line 10.

One irritating aspect of the ZX81 keyboard is that a space cannot be typed in under such circumstances as it BREAKs the program; however, the printout accompanying Listing 3.6 shows that a full stop serves to make the display of the name and address sufficiently clear. The program is in fact doing quite a lot more than just displaying data, which in normal circumstances would probably be loaded into the elements of a string array. It is also validating it to a limited extent. For example, the first item can consist of character codes 28 to 63, which is necessary as a National Insurance Number will contain both letters and numerals. The second item can only contain numbers (codes 28 to 37) as this is all that is required to enter the age - keyhits outside this range must be errors and are disregarded. In the same way, setting the maximum length of the "age" input to two characters eliminates a number of possible input errors.

SEARCHING AND SORTING

Most business programs will have to do a lot of searching through data, which in the case of the ZX81 will probably be stored in an array; it may also be necessary to sort the data before storing it in another array or perhaps sending it to the printer. Most of the methods of searching available to the ZX81 BASIC programmer carry bad news with them; they are either slow or greedy of memory space, and usually both. The simplest method is to use a FOR/NEXT loop to match up a search key with each of the elements of an array in succession; the trouble here, as has been pointed

Listing 3.6

```
  10 PRINT TAB 4;"NEW EMPLOYEE D
ETAILS"'''TAB 0;"N.I. NO.:";TAB
 20;"AGE:"'''TAB 0;"SURNAME:"'
'TAB 0;"FIRST NAME:"'''TAB 0;"
ADDRESS:"
  20 LET P$="6328090310"
  30 GOSUB 1000
  40 PRINT AT R,C;" ";
  50 LET P$="3728020325"
  60 GOSUB 1000
  70 PRINT AT R,C;" ";
  80 LET P$="6336150609"
  90 GOSUB 1000
 100 PRINT AT R,C;" ";
 110 LET P$="6336150512"
 120 GOSUB 1000
 130 PRINT AT R,C;" ";
 140 LET P$="6327311100"
 150 GOSUB 1000
 160 PRINT AT R,C;" ";
 200 COPY
 900 STOP
1000 LET B$=""
1002 LET Z$="■"
1005 SLOW
1007 LET R=VAL P$(7 TO 8)
1008 LET C=VAL P$(9 TO )
1010 PRINT AT R,C;
1012 IF LEN B$=VAL P$(5 TO 6) TH
EN RETURN
1015 GOSUB 1400
1020 IF CODE A$=118 THEN RETURN
1040 IF CODE A$=114 AND B$>"" TH
EN GOTO 1100
1050 IF CODE A$>=VAL P$(3 TO 4)
AND CODE A$<=VAL P$( TO 2) THEN
GOTO 1200
1060 GOTO 1010
1100 LET B$=B$( TO (LEN B$-1))
1101 PRINT AT R,C;" ";
1102 LET C=C-1
1110 GOTO 1010
1200 LET B$=B$+A$
1210 PRINT AT R,C;A$;
1220 LET C=C+1
1230 GOTO 1010
1400 LET Z$=("■" AND Z$=" ")+("
" AND Z$="■")
1420 PRINT AT R,C;Z$;
1425 LET A$=INKEY$
1427 IF A$="" THEN GOTO 1400
1430 RETURN
```

```
     NEW EMPLOYEE DETAILS

N.I. NO.: YC4472185 AGE: 43

SURNAME: HARRISON

FIRST NAME: CHRISTOPHER

ADDRESS:
3.ELM.ST.LONDON.W2
```

out, is the slowness of the FOR/NEXT loop in ZX81 BASIC. An alternative is to store a pointer to each item of data at the time that the data itself is stored; however, apart from the memory taken up by the stored pointers, it may be necessary to search through them instead. In the end, the way to speed up searches is to presort the data so that either the first item of data encountered by the search is the one we're looking for, or else we can be confident that there is a high probability of finding the item we want among the first few items examined.

That does not necessarily mean that the data must be sorted so that the items we are after are "at the top", such as, for example, being stored in the first few elements of an array. Provided that we know the principle that was used to do the sorting, we can use binary search methods to find the desired item in a small number of steps. In fact, we can predict the maximum possible number of steps in the search and so ensure in advance that excessive time will not be taken up by it. Binary search techniques will be covered at the end of this chapter. First, what is the best sorting method to use? In introductory works, when sorting is mentioned, one invariably finds a listing of a bubble-sort algorithm. This is on the whole the worst kind of sorting technique, as it is the slowest in the great majority of cases; the exceptional case is when the list is already mostly sorted.

Great controversy rages around sorting algorithms, but the one presented in Listing 3.7 is claimed by its proponents to be optimal in speed, and it doesn't take up too much memory either. The listing is shown as a test routine which you can use to find the best value for L. It's a shell-sort in which the program looks at two values in the array A, where the difference between their two subscripts is equal to L. Starting with the first element, it works its way up to the end, swapping values where necessary to produce a sequence of numbers from low (element 1) to high (element N, which is also defined by the user). It necessarily skips many numbers, so it then starts again with half the L value, and so on until L=1.

The idea is to avoid moving the same number more often than is essential. A bubble-sort would tend to get the numbers into the correct locations in the array in a large number of steps, adjusting the position of each number by only one subscript at a time. In fact an even better sort procedure is to have two values, say P1 and P2. Instead of comparing numbers at A(1) and A(L), such a routine compares numbers at A(P1) and A(P2), then increments P1 and decrements P2 and repeats the procedure. The trouble is that the listing then becomes quite long. This is not a problem from the point of view of memory space, but it has to be remembered that the ZX81 interpreter runs *very* slowly. In the listing shown, the actual sort routine is at lines 110-240, and although most of the lines are quite short, most of the time taken by the sort consists of the interpretation of the characters in the lines such as brackets, variable names and so on.

Listing 3.7

```
  1  SLOW
  3  PRINT "ARRAY SIZE? ";
  4  INPUT N
  5  PRINT N,,"SORT VALUE?   ";
  6  INPUT L
  7  PRINT L
  8  FAST
 10  DIM A(N)
 11  LET Z=2
 12  LET O=1
 15  RAND
 20  FOR X=O TO N
 30  LET A(X)=INT (RND*1000)
 35  NEXT X
 40  SLOW
 41  GOSUB 300
 45  SLOW
 50  PRINT TAB 0;"START"
 60  PAUSE 40
 70  FAST
110  LET L=INT (L/Z)
120  IF L<O THEN GOTO 245
130  LET T=0
140  FOR J=O TO N-L
150  IF A(J)<=A(J+L) THEN GOTO 2
00
160  LET S=A(J)
170  LET A(J)=A(J+L)
180  LET A(J+L)=S
190  LET T=O
200  NEXT J
230  IF T THEN GOTO 130
240  GOTO 110
245  SLOW
247  PRINT "DONE"
250  GOSUB 300
255  FAST
260  STOP
300  FOR X=O TO N
310  PRINT A(X);"   ";
320  NEXT X
330  RETURN
```

This slowness can be shown by setting the array size to 50 and the value of L as 32. Despite the fact that the array has been filled with integers, which should be very slightly faster to deal with than values with decimal components, the routine takes around 24 seconds to sort the random values into ascending order. Choosing an L value of 16 reduces this to about 16 seconds, and indeed one should experiment with different L values once the array size is known, as this can bring substantial improvements. Nevertheless, sorting 1000 numbers or so is always going to be a time-consuming task.

The best advice to the business user, therefore, is to keep arrays of data to a modest size whenever it is going to be necessary to sort them. Once the sorting has been done, a binary search procedure is probably the best way to get at a particular item of data. An example of such an algorithm can be seen in Listing 3.8, where some lines have been tacked on to Listing 3.7. The value of

K has previously been set at 700 (any value can be chosen) and this has been inserted into the sorted array. Lines 275 and 280 will look at the value stored in A(Q) and, if that value is greater than K, then half the array can be disregarded from that point on (the top half in this case, elements 21-40 if a 40-element array is being used). Similarly, the bottom half can be disregarded if the value is less than K.

Listing 3.8

```
260 LET O=N/2
262 LET OO=N
265 REM
270 IF A(O)=K THEN GOTO 500
275 IF A(O)<K THEN GOSUB 700
280 IF A(O)>K THEN GOSUB 600
285 GOTO 265
300 FOR X=0 TO N
310 PRINT A(X);"   ";
320 NEXT X
330 RETURN
500 PRINT "FOUND AT ";INT O
510 STOP
600 LET OO=O
610 LET O=O/2
620 RETURN
700 LET O=O+(OO-O)/2
710 RETURN
```

The remainder of the array that still needs to be examined is divided in half once again, by the subroutine at 600 or 700, and another examination of the resulting A(Q) value is made. This procedure of successive subdivision quickly reduces the area within the array that needs to be searched, and in the example of a 40-element array, it only takes a split-second to find where K is stored and jump to line 500 where the answer is printed on the screen. The great advantage of the binary search is that the search-time does not increase linearly with the size of the array, so that it will still take only a very short time to find a value in a thousand-element array; but, as mentioned above, getting the values in that array sorted quickly is a trickier problem.

Chapter 4 Supergraphics

In the early days of microcomputing, users were instructed by manuals to use PLOT statements (or their equivalent) to create graphic displays, not only for graphs but also for games. Although some programs are found that use such methods, authors of game software generally don't implement their programs in this way nowadays, but use string-printing techniques instead. This is the procedure known as "supergraphics", and on most computers it takes two distinct forms. One method is to build up a string literal by using character codes and concatenating them so that they are embedded in a long string; the other is to set up a dummy string and POKE the appropriate codes into it.

The distinction is of less importance on the ZX81, where nearly all the graphics characters available to the BASIC programmer are accessible from the keyboard, so that the string can be typed in just like a BASIC program line of keywords and numeric values. Normally the ZX81 programmer will set up a string which already contains the characters he wants, typed in from the keyboard. There are still occasions when concatenation is needed, as in Listing 4.1. This shows a method of creating a black screen to use

Listing 4.1

```
  2 FAST
  4 LET K=704
  9 DIM B$(K)
 10 FOR X=1 TO K
 22 LET B$(X)="■"
 30 NEXT X
190 CLS
195 SLOW
200 PRINT B$
300 CLS
400 PRINT B$
```

as a background for the display of white-on-black characters. If a string of 704 black squares is built up in advance of the main program, it can be printed on the screen very quickly (though not exactly instantly) even in SLOW mode. The CLS statement is even quicker, so this is a way of making the screen flash between black and white, as is done in the program in the Listing.

It takes six seconds for the 704 characters to be built up in the A$ array, and the normal method of dealing with such delays in games programs is to print the rules of the game on the screen

while this initialising process is going on; once all the string variables have been set up, the game can be played repeatedly without further delays. Note that the Listing stores the characters in an array rather than a string variable. There is not a lot of difference between the two types of string-storage in ZX81 BASIC, but the reason for doing it this way is that the initial delay is much shorter. This is because the interpreter sets aside 704 bytes for storage of the string all in one operation, as it executes line 9. If, instead, we did it by setting up a null string:

```
9      LET B$=""
```

and then built up this string variable by concatenation:

```
22     LET B$=B$+"(black square)"
```

the interpreter would have to extend the length of the variable table by one byte on each iteration of the loop, since it "wouldn't know" what the final variable-length was going to be. The additional housekeeping that the operating system has to do, when the length of the variable table changes, causes an additional delay of at least another five seconds.

Whichever method of building up a string is chosen, the programmer will want to embed cursor-control codes in the string to move the screen-cursor around in between printing parts of the string on the screen. Sadly, this can't be done on the ZX81. Although there are character-codes representing the four arrow-symbols on the keyboard, as well as code 118 decimal for NEWLINE (carriage-return), these take effect only during editing operations or when moving the program cursor down a listing on the screen. Trying to use them in a string-literal just produces a question-mark. This, together with the slow execution speed of ZX81 BASIC in SLOW mode (equivalent to about ½ MHz), severely limits graphics applications of the computer which require speedy display of images. In the end the answer is to use a bit of machine-code to speed things up; information on various ways of doing this will be given in later chapters.

The absence of cursor-control during printing on the screen causes problems which are illustrated by the various examples in Listing 4.2. This is an extension of 4.1, and maintains a black background while showing the speed at which the same graphic image (a grey rectangle) can be built up using different coding techniques. In the first example, lines 500-510, the image is contained in a single print-literal; this is the fastest of the routines, but unfortunately it blanks out the last part of the first line on which the image is printed, all of the second line and the first part of the third line. We could get over this in the example given, by modifying the print-literal so that it printed black squares instead of white spaces, but of course this is of no help if we want to print something else in the area surrounding the image and not have it erased every time the image is printed.

Listing 4.2

```
    0  FAST
   40  LET K=704
   90  DIM B$(K)
  100  FOR X=1 TO K
  200  LET B$(X)="■"
  300  NEXT X
 1000  CLS
 1100  SLOW
 2000  PRINT B$
 3000  CLS
 4000  PRINT B$
 4100  PAUSE 100
 5000  PRINT AT 0,0;
 5100  PRINT "■■■■■■■■
                ■■■■■■■■";
  540  PAUSE 100
 5500  PRINT AT 0,0;B$
 5600  PAUSE 100
 5800  PRINT AT 0,6;"■■■■■■■■";AT 9,6
;"■■■■■■■■";AT 10,6;"■■■■■■■■";
  640  PAUSE 100
 6500  PRINT AT 0,0;B$
 6600  PAUSE 100
 7000  PRINT AT 0,0;
 7100  LET P=6
 7200  PRINT TAB P;"■■■■■■■■";TAB P;"
■■■■■■■■";TAB P;"■■■■■■■■";
  740  PAUSE 100
 7500  PRINT AT 0,0;B$
 7600  PAUSE 100
 8000  DIM Z$(3,5)
 8100  LET Z$(1)="■■■■■■■■"
 8200  LET Z$(2)="■■■■■■■■"
 8300  LET Z$(3)="■■■■■■■■"
 8350  PRINT AT 0,0;
 8400  FOR G=1 TO 3
 8500  PRINT TAB P;Z$(G);
 8600  NEXT G
 8700  PAUSE 100
 8800  PRINT AT 0,0;B$
 1000  FAST
```

To solve this problem, the second routine (line 600) uses PRINT AT. However, the routine is perceptibly slower in building up the image. An improvement is gained in the routine at 700-720 by setting P equal to the column number and also by using TAB. Apart from saving memory, there are fewer numeric literals in the line for the interpreter to decode into locations within the Display File, so it is that much faster.

The routine at 800-860 seems to be a step backwards. Apart from being much longer, it is noticeably slower. However, imagine what would happen if the image we wanted to print stretched over (say) twelve lines of the screen and we wanted to change some parts of it from time to time before printing the new version. The FOR/NEXT loop, used to print each element of the string array in turn, would then be the best way to do it.

One often comes across routines which POKE byte values to the Display File in order to build up screen images. These seem to be

based on a misconception about the relative speed of different routines within the BASIC interpreter. Although POKE is quite a fast routine, it requires two numeric arguments, the memory address and the byte value.

The first argument can be worked out by the program, by finding the address of the start of the Display File in locations 16396/7, but it then has to do some calculations to get from that to the memory address where the value is to be stored; the value itself will normally be quoted as a numeric literal in the program and the interpreter will have to convert it from decimal to binary. As a result of all this, a routine to build up images in this way will be quite slow and offers no compensating advantages over the methods described here. It would be quite different if the values were being loaded into the memory locations by a machine-code program, but that is another matter.

ANIMATION

We're still a long way from the arcades; what is the best way of getting our images to move around the screen? There are two problems: blanking out the current image before printing it in a different location, and getting it to move as quickly as possible. Listing 4.3 attempts to cope with both problems. As you can see,

Listing 4.3

```
  1 GOTO 100
 20 PRINT AT R,0;
 25 RETURN
 30 PRINT TAB P;"
AB P;"    ";TAB P;"
";TAB P;"    ";
 40 RETURN
 50 PRINT TAB P;A$;TAB P;A$;TAB
P;A$;TAB P;A$;
 70 RETURN
100 REM MAIN ROUTINE
102 LET A$="                "
103 LET Z=20
104 LET Y=1
106 LET J=30
107 LET V=50
109 SLOW
110 LET R=12
115 FOR P=Z TO Y STEP -Y
117 GOSUB Z
120 GOSUB J
125 GOSUB Z
130 GOSUB V
140 NEXT P
200 CLS
310 PRINT AT 18,6;
320 LET P=8
330 GOSUB 30
340 FOR Q=Y TO 12
350 SCROLL
400 NEXT Q
```

it uses my impression of a sideways view of the Starship Enterprise; however, I make no claims to artistic skill. The program uses a number of devices to increase speed: one of them is to place the subroutines at the beginning of the program so that they will be accessed faster, but in this particular program you could relocate lines 20, 30 and 50 to place them within the main screen-display loop in 115-140. It depends whether you will need to do things like erasing the picture currently on the screen (this is what line 50 does) at several places within the program.

The Listing also uses short variable names for line-numbers as objects of GOSUBs; this increases speed slightly. The effect is that the fairly complex shape of the Enterprise moves across the screen from right to left. Each previous image is wiped out by an eleven-byte string of spaces repeated four times. One disadvantage of this is that the area really occupied by the Starship is slightly more than it appears, being a 44-byte rectangle on the screen, and anything else that appears within that rectangle will be wiped out by line 50. You could alter this by using four distinct strings of different lengths to achieve the erasure of the four strings making up the image, and indeed you could eliminate the leading spaces which are included in the string-literals quoted in line 30. Then, however, you would need a number of numeric variables to act as different PRINT TAB position, whereas the existing technique lets you use just one (P). The longer the variable table gets, the longer the ZX81 will take to retrieve a variable value from it.

If you type in and run the program as shown, there is a final display of a different kind; the Starship moves from the bottom of the screen to the top. This time there is no impression of a picture constantly being erased and redrawn, and the display is altogether smoother. This is achieved by using SCROLL, which relocates all the screen lines by moving them one line up, somewhat faster than can be done using a BASIC routine. The SCROLL statement offers the opportunity for fast graphics of a particular type, best exemplified by the game "Meteor Storm". An example of a program of this kind is given in Listing 4.4, which is only a partial implementation of the game. Plus-signs representing the meteor shower are printed in random positions on the bottom row of the screen and scrolled up in an infinite loop. This means that the spaceship, represented rather unimaginatively by a black square, must continually be printed again in its fixed position at the top of the screen. By hitting the keys at either end of the bottom row of the keyboard, the ship can, however, be made to move left and right to dodge the meteors. The next step is to give the ship a laser cannon to destroy meteors directly in its path. This can certainly be done, but as soon as one adds the necessary logic to the program, it begins to slow down too much. SCROLL is a good way of moving the entire screen display, but the slow execution speed of ZX81 BASIC represents a barrier to the development of sophisticated graphics which can't be overcome. The best advice to those who do not want to move on to machine-code in order to sidestep the problem is to restrict one's ambitions to small displays.

Listing 4.4

```
   2 FAST
   4 LET R=16396
   6 LET P=16397
   8 LET V=256
  10 LET O=0
  12 LET N=1
  14 LET L=22
  16 LET J=31
  18 LET Q=100
  20 LET F=1000
  30 RAND
  32 LET D$="PEEK (Z+1+PEEK R+PE
EK P*V)"
  40 LET U$="▀"
  50 FAST
  60 CLS
  70 LET Z=15
  90 SLOW
 100 PRINT AT O,Z;U$;
 130 PRINT AT L,RND*J;"+";
 140 SCROLL
 152 IF VAL D$=L THEN GOTO F
 200 IF INKEY$="Z" THEN LET Z=Z-
N
 210 IF INKEY$="." THEN LET Z=Z+
N
 220 IF Z<O THEN LET Z=O
 230 IF Z>J THEN LET Z=J
 260 GOTO 9
1000 IF Z<2 THEN LET Z=2
1002 IF Z>29 THEN LET Z=29
1005 PRINT AT 0,Z-1;"▄▀▄";AT 1,Z
-2;"▄▀▀▄▀▀▄"
1010 PRINT AT 15,5;"ANOTHER GAME
 ?"
1020 IF INKEY$="Y" THEN GOTO 50
1030 IF INKEY$<>"N" THEN GOTO 10
20
1040 FAST
```

Listing 4.5

```
  10 LET A$="▀▄      ▄▀"
  20 LET B$="  ▀▄▄▄▄▀  "
  30 LET C$="   ▀  ▀   "
  40 LET D$="  ▄    ▄  "
  44 LET Z$="         "
  46 LET P=12
  47 LET O=14
  50 SLOW
  90 FOR X=25 TO 1 STEP -1
 120 PRINT AT P,X;A$;TAB X;B$;TA
B X;C$;
 134 LET K=RND*PI
 160 PRINT AT O,X;D$;
 170 LET K=RND*PI
 190 PRINT AT P,X;Z$;TAB X;Z$;TA
B X;Z$;
 200 NEXT X
```

A surprising amount can be achieved within these limits. Listing 4.5 is a rather entertaining example of animation which uses a technique familiar to all those who have worked in the field of cartoon animation: one part of the image is unaltered but is printed in different screen areas in succession; meanwhile, other parts of the image alternate between two different versions, giving the impression of a running dachshund.

The program is structured so as to minimise the amount of printing-and-erasing that has to be done. The top two strings produce the head, tail and body of the dog, and C$ portrays straightened legs. It's then necessary to insert a short delay into the program to allow this visual image to register with the viewer. PAUSE would be the obvious choice here but, unfortunately, it produces a screen flicker which we want to avoid. It turns out that a FOR/NEXT loop takes a little too long for the interpreter to evaluate even if it is set to count FOR D=1 TO 1, so instead we use, in lines 134 and 170, the dummy expression which is shown in the Listing, and which is evaluated by the interpreter after just the delay period we need. A slightly longer delay can be produced by substituting a numeric literal for PI. We need the delay after D$ is printed as this represents the legs at an angle, and again the viewer needs a split-second to take it in.

Very little visual information is in fact produced by the entire program, and yet it's sufficient to convey the right perception in the user, who supplies most of the missing information in the act of "recognising" the view of a running dog. Similar effects can be produced by careful selection of print-string symbols.

SCREEN DRAWING

All the images used in the Listings in this chapter could of course be produced by using PLOT, but if you try to recreate even the smallest of these by this means, you will soon see that it's far too slow for most practical purposes. There is, however, a very useful application for PLOT in the planning stages of any graphics program that will need a complex figure, to be built up with graphics characters. The idea is that it's quite tedious to work out the combination of graphics characters needed to make up a given shape; it has to be done on a trial and error basis, and editing long strings to change one or two characters is time-consuming. One would like to just draw the figure on the screen, and then have a routine which automatically works out the equivalent byte-values for the graphics strings. Naturally one wants the best resolution possible, which will mean using all the available graphics symbols, and yet it seems that the only way to build up a picture at maximum resolution is to use a program which PLOTs pixels onto the screen.

In fact it's possible to build up the image using PLOT and still have the program work out the byte values for the graphics charac-

Listing 4.6

```
   10 FAST
   50 DIM A(4,6)
   60 LET A$="PEEK (Z+PEEK 16396+
PEEK 16397*256)"
   70 LET Z=1
   80 LET R=0
   90 LET C=43
  100 LET PLOT=1
  200 SLOW
  210 IF INKEY$="U" THEN LET C=C+
1
  220 IF INKEY$="D" THEN LET C=C-
1
  230 IF INKEY$="R" THEN LET R=R+
1
  240 IF INKEY$="L" THEN LET R=R-
1
  250 IF INKEY$="P" THEN LET PLOT
=1
  260 IF INKEY$="X" THEN LET PLOT
=0
  270 IF INKEY$="Z" THEN GOTO 200
0
  280 IF R>11 THEN LET R=11
  290 IF R<0 THEN LET R=0
  300 IF C>43 THEN LET C=43
  310 IF C<36 THEN LET C=36
  350 IF PLOT THEN GOSUB 3000
  360 IF NOT PLOT THEN GOSUB 4000
  400 GOTO 210
2000 PRINT AT 3,20;"DONE";
2005 LET PR=10
2010 LET PC=0
2050 FOR R=1 TO 4
2060 FOR C=1 TO 6
2070 LET A(R,C)=VAL A$
2080 LET Z=Z+1
2090 PRINT AT PR,PC;CHR$ A(R,C);
A(R,C);" ";
2100 LET PC=PC+5
2110 NEXT C
2115 LET PC=0
2120 LET Z=Z+27
2130 LET PR=PR+2
2140 NEXT R
2200 STOP
3000 PLOT R,C
3010 UNPLOT R,C
3020 PLOT R,C
3030 RETURN
4000 UNPLOT R,C
4010 PLOT R,C
4020 UNPLOT R,C
4030 RETURN
```

```
                              DONE

 3   132   0     0    7   132
135  133   0     0    5   133
133   2    3     3    7   3
 6    3    1     2    1   0
```

ters. Listing 4.6 shows how to do this. The essential idea is that a one-byte graphics character is only the equivalent of a combination of four PLOTted or UNPLOTted points, so far as the ZX81 is concerned. For example, suppose you executed:

```
10     PRINT AT 0,0;"(black square)"
```

As a result of this, the memory location holding the first byte of the Display File would contain the value 128. If you were to clear the screen and then execute:

```
10     PLOT 0,43
```

a pixel would appear top-left on the screen, and the memory location holding the start of the Display File would now contain the value 1. This isn't a special byte-code for PLOT values, it's just the character code for a graphics character made up of one black pixel top left and white space for the other three pixels. Therefore, you could PLOT the three adjacent pixels to the one you had already PLOTted, and then the memory location in the Display File would once again contain the value 128, since a square of four PLOTted points is equivalent to the black square graphics character.

Of course, it can get more complicated if a black square on the screen is really made up from a black rectangular graphics character on one line, and another rectangular character in the same column of the next line down. Then you have to look up the two different characters and put them into the appropriate one-line string literals. Listing 4.6 is designed to avoid this procedure, by allowing you to draw any image you like using PLOT and then working out the equivalent in terms of lines of graphics characters. A$ is used to hold the string representation of a function which will find the memory location of a given byte in the Display File; lines 200-400 contain the drawing routine. It incorporates a repeat-key effect, so that you can draw a line from left to right just by holding down the R key. The keys L, U and D handle movement left, up and down. If you want to erase part of the picture, hitting X will put the routine in UNPLOT mode, and of course this key is also used for moving the "paintbrush" to another part of the screen without leaving a black trail behind. Hitting P will put the brush back in PLOT mode.

In the listing, the picture is built up in the top left part of the screen, allowing an 8 by 12 pixel grid; it could be larger, but you need room on the screen to display the equivalent graphics characters and byte values. A small refinement in the program is that the current location of the brush on the screen is indicated by flashing that pixel on and off, making things much easier.

When the picture is complete, the user hits the Z key. The routine will then fill up the array A with the byte values representing the 24 graphics characters that make up the pixture. It does this by PEEKing directly into the Display File, using A$ to

calculate the locations to examine. At the same time it prints the graphics character, immediately followed by the equivalent byte value, arranging this material in a 4 by 6 chart on the screen.

It can be seen that the Listing as given doesn't actually need an array, but the advantage of storing the values in an array is that one can clear the screen and draw another picture without losing the information about the first one. For example, if a third dimension of ten elements is added to A, ten pictures can be built up in succession, and then each one can be displayed on the screen by writing a routine to print the CHR$ equivalent of the values held on each "page" of the array.

The one limitation of this program is that it can't handle grey characters, but of course the Sinclair character set is also limited; of the various permutations of four grey pixels within a graphics character-square that can be worked out, only a small number are actually available in the character set and therefore printable on the screen.

By combining all the techniques described in this chapter, the BASIC programmer can achieve effective graphics animation at a modest level of sophistication; and that's all he can do with a ZX81. The limiting factor is the speed of the BASIC interpreter and, until BASIC compilers become available for the machine, the programmer must turn to machine code for more elaborate programs. It is worth pointing out that there are two possible approaches to the use of machine code for this purpose. If we once again consider Listing 4.3, and compare it with the effect it produces when the program is typed in and run, it becomes quite clear that some fragments of the BASIC program are already adequate to our requirements, and it's only a few lines that are slowing down execution to the point that no further enhancements can be added to the routine without ruining the animation effect. One problem is that the four-line image does not appear instantly, but takes a perceptible split-second to build up. On the other hand, one notices no problem with the part of the program which erases this image. As a matter of fact it takes about the same length of time for the erasing to take place, but because of the way our perceptual systems work, we tend not to notice it so much.

The PRINT AT statement is also quite slow; this is not so apparent from the program, but soon becomes obvious when one experiments with it. The delay is in that part of the ROM routine which calculates the new position for the screen cursor and stores equivalent values in the operating system extension in low RAM.

Finally, the FOR/NEXT loop is itself slow, particularly when it has to count backwards as in Listing 4.3. Other parts of the program, such as the GOSUB calls, do not add any noticeable delay and there would be no particular point in writing a machine-code equivalent for them unless we had in mind a very ambitious programming project.

We could therefore consider writing about three short machine-code programs, each one intended to replace one of the lines in

the program. Each would have to be accessed with its own USR call; in some cases it would be necessary for the result of the routine (such as a new PRINT AT location for the cursor) to be passed back to BASIC, which is something the USR statement can do. One would end up with a thoroughly hybrid program, alternating between BASIC lines and machine-code fragments.

The attraction of this technique is that the minimum of machine-coding has to be done. However, the limiting factor is the execution time of the USR statement; this is still part of BASIC, and effectively adds a fixed time-overhead to each machine-code routine that is called directly from BASIC. The other problem is that the programmer will have to be very careful that the routines he uses carry out exactly the same housekeeping tasks as are implemented by the BASIC statements they replace. As an example of what is meant by housekeeping, consider the PRINT AT column and line numbers which are stored in locations 16441 and 16442. Nothing is easier than to write new values to these locations during a machine-code program, in order to move the cursor around the screen. However, this is not all that the BASIC PRINT AT statement does; it also has to adjust the values in 16398 and 16399, which represent the memory location within the Display File into which the next PRINTed character-value should be loaded. Working out one set of values from the other is not entirely straightforward, but if the programmer does not emulate BASIC in this way, strange things may happen to his program during subsequent execution. Most of these problems can be avoided by staying in machine code until all the necessary visual effects have been produced, and only then returning to BASIC; the machine-code routine will then be longer, but the programmer will have to spend far less time worrying about the way the operating system works. Examples of both approaches to the problem will be given later, but in most cases I would strongly recommend against attempts to replace isolated BASIC statements with machine-code routines on a one-for-one basis; the way in which the BASIC interpreter works is much more complex than this.

Chapter 5 Artificial Intelligence: Simulating intelligence

Artificial Intelligence programming plays a large and increasingly significant part in the application of computers to commercial and leisure activities. There are a number of reasons for this. A great deal can be achieved with a sophisticated number cruncher, such as the large computers used in scientific and technical processing, but a machine of this kind is still limited to the sequential treatment of a very large number of pieces of data, and will have to be told by its program exactly what to do with each one. It would be much more interesting if we could feed a computer all the information we could lay our hands on, sit back and ask it the answer to Life, the Universe and everything.

What prevents us from doing this? For one thing, the computer doesn't understand English, and we must communicate with it in one of the available computer languages, none of which seems to have made provision for enquiries of this kind. Nor is the machine designed to mull over a large number of pieces of data, looking for interesting connections and analogies; the fifth generation of computers now being planned may include parallel processing circuitry which makes this idea a little more feasible. Finally, it seems that an enormous amount of effort, represented by extremely long programs, must be put into an AI project in order to produce even the simplest result. Such a program may allow the computer to give the impression of competence within a very restricted field, but the "payoff" is so limited compared to the input that we feel that, surely, this can't be the way intelligence works. Further progress is hampered by the fact that we really don't know how human intelligence works; the more we consider the matter, the more our ignorance becomes apparent.

The question of natural-language input to computers is central to AI and will be considered in Chapter 6. The kind of computer that is currently available to the home user is hardly powerful enough to undertake the sophisticated data-manipulation operations that are involved in any program that "searches its memory" in an intelligent way, and can come up with unexpected answers. However, there are various ways in which a computer can be programmed to give a variety of interesting responses, all of them relevant, after a comparatively small effort has been made by the programmer. Successful programs of this kind are nearly always games.

Consider the definition of a good game. The rules are unambiguous and not too extensive. The playing area and playing pieces, if any, are usually not too numerous and are known to both players

in advance. The outcome of the game depends on the skill of the players in operating within the rules, and although all the facts about the mechanics of the game are known, the result of play is very difficult to predict merely from a consideration of the strategies being used by the players; it's necessary actually to play the game to find out who will win, and the number of possible different outcomes of the game should be extremely large.

One approach that has been taken by AI researchers is to write a program that plays a particular game well, preferably to an unbeatable standard. The ZX81 programmer who is interested in the problems of AI may well want to follow this route. He should, however, be careful about the choice of game. Probably the simplest game that is commonly implemented on microcomputers is Noughts and Crosses. There are two ways of doing this; one is to work out algorithms corresponding to optimal strategy in the game, which is not too difficult, and feed them into the program. The other is to make the computer choose moves at random, but only after discarding moves that it has previously tried in losing games. Eventually all the bad responses are discovered and entered into the database, although it can take quite a few games before this happens.

Not a lot about AI techniques can be learnt from such an exercise. The "brute force" approach of trying all the moves, good and bad, does not correspond at all to human practice and is feasible in this case only because Noughts and Crosses is such a simple game; to attempt the same approach with a more complex and interesting game will lead to an enormous increase in the permutations of moves that must be tried before intelligent play can be achieved by the computer. Working out the algorithms is a programming exercise for someone who wants to develop his understanding of a particular computer language, but this is not the same as discovering how intelligence works. After all, the programmer already knew how to play Noughts and Crosses well, but he had to make the effort of translating his knowledge into a computer program; having done so, his skill at the game is unlikely to be any greater than it was before.

At the other extreme, there are already several Chess programs for the ZX81, and there is no reason why one should not write another, or try something less ambitious like Draughts. Even a Draughts program is still a major project, and many hundreds of man-hours should be set aside for it. The programmer should also consider what he will achieve as a result. Ability to play Draughts well is after all a very small aspect of an individual's total intellectual capabilities. It is an ability not an aptitude, that is, it represents the result of playing many games of Draughts and inferring correct strategy from this experience. The programmer is smart enough to carry out the same procedure with any other game that he wishes to play well, but his smartness is not contained in the strategy for Draughts; this strategy can be programmed into a computer as a Draughts program, but the computer will not thereby become any smarter, and the program will not enable it to play any other game.

The program for the ZX81 which makes up the rest of this chapter is the result of taking a different approach to the problem. It's a game, but of an unusual kind. The computer controls one playing piece and the player another, the object being to destroy the opponent's piece. The player, however, does not make moves during play; instead, he must write down his entire strategy for the game in advance, in the form of a sequence of instructions to the playing piece. This is in fact a program, written in a very simple language devised specially for this particular game. The computer similarly manipulates its piece by following a program chosen in advance.

Acknowledgement should be made to the inspiration for the game, called *Arena*. The American company Muse Software released a program for the Apple computer in 1981, called *Robotwar*. I have never seen the game, and indeed it has attracted little attention in the UK, but the idea for *Arena* came from reviews of *Robotwar*. The point should be made that *Arena* is a much simpler game, and is intended as an example of a particular approach to games programming on which the reader will probably be able to improve.

Why choose this approach? *Arena* was not very difficult to write, but a number of interesting conclusions emerge from it which seem to repay the effort. The rules are so simple that the player ought to know in advance exactly what will happen, but this is not the case in practice. The player can study everything there is to be known about the "strategy" that the computer will use, since this is contained in the program that will govern the actions of the computer's playing piece, and the player has access to this "program". The player himself can write any program he likes to govern the actions of his piece, but it turns out to be remarkably difficult to visualise the effect of a given program and thus deduce the optimal, winning program.

There is also the consideration that this level of programming complexity is about right for the 16K ZX81, assuming that we are writing in BASIC; anything more complex would probably run too slowly to be practicable, or take up too much space. At this level, however, there are many games of the *Arena/Robotwar* type that could be written, and each should offer some insights into the abilities and limitations of computers in the field of AI programming.

ARENA

Imagine a futuristic battleground in which robots do the actual fighting, following programs built into them by their makers. Such programs cannot cater to every contingency, but must seek to give the maximum amount of combat effectiveness in the various general situations that may arise. The authors of the programs have to envisage possible battle conditions and construct a series of instructions which tend to give a robot - depicted in *Arena* as a kind of ponderous tank - the best chance of destroying its enemy.

No program can be endless, so the set of instructions will loop back on itself sooner or later. They must, therefore, be devised in such a way that the robot is not trapped in a futile sequence of actions that does not lead to tracking down and eliminating its opponent.

In the game *Arena*, the battleground is a 10 by 15 grid on the screen, and there are two robot tanks, one controlled by the player, one by the Enemy. Turns alternate, and at each Turn a robot can do one of five things:

MV Move one square in the direction it is facing. If this brings it to the edge of the grid, the program automatically reverses it to face in the opposite direction.

RL Rotate left. The robot turns 90 degrees anticlockwise.

RR Rotate right. The robot turns 90 degrees clockwise.

FR Fire. The robot has a front-facing cannon with a range of five squares. Firing it causes a black blob to move away from the front of the robot tank, and if it coincides with the enemy tank, this scores one Hit.

FD Find. This causes the robot to turn towards the enemy tank. In practice it can only turn in the general direction since it is limited to orthogonal facing.

This is not, however, an "interactive" game in which the player inputs his moves Turn by Turn. Instead, he must use a "program" consisting of 22 commands, each of which must be one of the five actions given above. The Enemy robot will have a similar program, also of 22 commands. Once the game starts, both machines relentlessly execute their programs, action by action, returning to the beginning of the sequence as they reach the end. The game continues until one robot has destroyed the other. There are two ways of doing this. If a robot "rams" its opponent, it automatically destroys it, winning outright. Otherwise, if a robot receives two Hits from its Enemy's cannon, it is eliminated and the opponent wins.

The point of the game, of course, is to devise the most effective program for eliminating any opponent. The set of commands is about as elementary as it could be, and it shouldn't be difficult to work out the optimum program. After all, the potential of the game is limited by the geometrical characteristics of the *Arena* and the restricted abilities of the robots. There are, however, many millions of possible programs and it turns out to be not at all easy to work out the best one. It should be possible to deduce the optimum program in the abstract, by reference to some standard principles of game theory, but I know of no way of doing this; in practice, one must devise a program, test it by running it against a variety of opponents, and then rework it. The optimum program will, of course, be the one that not only wins the game for its robot, but does so in the shortest average period of time against all opposition.

If you type in and run *Arena*, it will begin by initialising a number of variables and then displaying a menu of choices. There is a Player-program built into the game - it's in line 2510 - and you can use this to control your own robot without having to devise a program of your own. In that case you have to select one of four possible opponents. Programs for these opponents are also in the listing, and have been worked out to show the effectiveness of various simple combat strategies. Killer uses a lot of Fire commands, trying to spray the whole Arena with cannon-shot, and periodically Rotating to make sure of covering all angles. The Player-program can usually beat him although it may take a while.

Seeker emphasises use of the Find command, usually following this with a Fire instruction or two. This ought to work rather well, but it can lead to some very protracted "end-games" against the Player program. The reason is the inadequacy of the Find algorithm, which checks the relative locations of the two units on the X axis and then on the Y axis and changes the facing of the Finding unit accordingly. If the enemy unit is diagonally removed from the Finding unit, this will return only an approximate vector and Seeker may tend to run past the Player robot without noticing. The Player program is similar to Seeker, but includes a few Move-Find-Fire sequences intended to overcome the inadequacy of the Find mechanism.

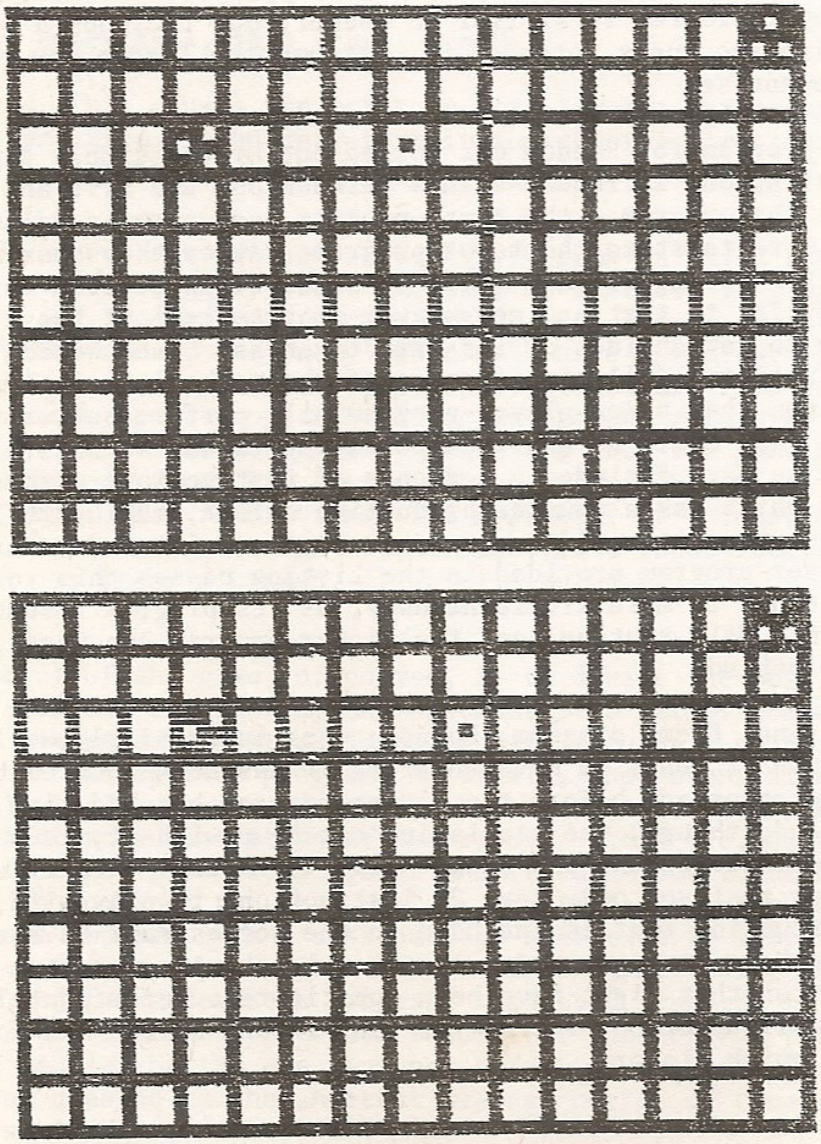
The program for Random was worked out by the simple expedient of printing out 22 random values between one and five and using them as the program - the *Arena*-program uses numeric values from one to five to store the robot-programs rather than storing the two-letter abbreviations. This may sound flippant, but in fact it is essential to test any new player-program against the Random program to get an idea of its effectiveness. Since we don't understand (yet) the full significance of the mechanics of *Arena*, we don't know that a new player-program will perform better than the results that could be achieved purely by random actions. It could be that we have included a sequence of instructions within our program which has a counter-productive effect, making it less efficient than it would be by random action alone. Fortunately, the player-program provided in the Listing passes this test; Random tends to make little headway, as its program resembles a Drunkard's Walk routine, and the player-program can usually catch it from behind.

The final Enemy program, Runner, does surprisingly well; this is because it tends to move on after it has been located by the Player-program and before the Player-program has had time to fire. In the end, though, the statistics catch up with it; it has very few Fire instructions and so is a bit toothless. This illustrates the point that you only have 22 instructions to play with, and so any instruction that is included in the series must be more "cost-effective" in its particular location in the program than any other instruction that might have been substituted. Unfortunately, Runner occasionally overruns the player-tank in one of its mad dashes and earns a quick victory.

If you choose one of the Enemies and run it against the built-in player program, your unit appears on the screen first, followed by the Enemy. However, the Enemy unit executes its first command first, followed by your first command, and so on. Initial dispositions are random. If you want to interrupt the game, which can sometimes take a long time to reach its conclusion, hit any key and hold it down; you will be returned to the menu.

If you type in your own program, enter 22 instructions using the two-letter abbreviations for actions listed above (invalid input will be rejected). The *Arena* program will then auto-run your program against Killer, but you can of course interrupt this and choose another opponent. You can preserve your program, which has overwritten the built-in player-program in the first row of the E$ array, by hitting Break and entering GOTO 9000. This will SAVE the whole *Arena* program to tape, including the variable table with your program. Reloading *Arena* takes about 3¾ minutes, as it's about 10K, and it will auto-run on reload.

Listing 5.1



```
   5 SLOW
  10 PRINT ,,,,,,,TAB 12;"ARENA",
,,TAB 13;"BY";TAB 8;"MIKE COSTEL
LO";TAB 10;"AND ENJAY",,,,,,,,,TA
B 7;"PLEASE STAND BY"
  15 PAUSE 80
  20 FAST
  30 LET INITIALISE=2000
  32 LET STORE=3000
  34 LET MENU=7000
  36 LET COMMAND=8000
  38 LET NEWPROG=8500
  44 LET PLACE=5000
  46 LET WIPE=6200
  48 LET DISP=6000
  55 DIM E$(5,2)
  65 DIM D(23,5)
  70 LET X$="(INT (RND*15)*4)+2"
  72 LET Y$="(INT (RND*10)*4)+5"
  75 GOSUB INITIALISE
  80 LET PLAYER=1
  82 LET PPTR=1
  83 LET EPTR=PPTR
  84 LET PTR=PPTR
  86 LET WIPEOUT=0
  88 LET PDEF=2
  90 LET FDEF=2
 200 CLS
 203 GOSUB MENU
 205 IF X=29 THEN GOTO NEWPROG
 206 LET ENEMY=X-28
 207 LET PHASING=ENEMY
 208 CLS
 210 PRINT B$;
 250 GOSUB PLACE
 300 REM THIS IS START OF MAIN L
OOP
 310 GOSUB COMMAND
 330 IF INKEY$<>"" THEN GOTO 200
 350 IF WIPEOUT THEN GOTO 1500
 400 LET PHASING=(ENEMY AND PHAS
ING=PLAYER)+(PLAYER AND PHASING=
ENEMY)
 410 LET PTR=(PPTR AND PHASING=1
)+(EPTR AND PHASING<>1)
 415 LET DIR=(LOCDIR AND PHASING
=PLAYER)+(FOEDIR AND PHASING=ENE
MY)
 420 LET XP=(XLOC AND PHASING=PL
AYER)+(XFOE AND PHASING=ENEMY)
 430 LET YP=(YLOC AND PHASING=PL
AYER)+(YFOE AND PHASING=ENEMY)
1000 GOTO 300
1001 REM END OF LOOP
1500 REM ******WIPEOUT*******
1510 CLS
1520 PRINT "END OF GAME"
1530 PRINT
1600 IF WIPEOUT=2 THEN GOTO 1700
1610 PRINT "PLAYER UNIT DESTROYE
D"
1620 PAUSE 50
1630 GOTO 80
1700 PRINT "ENEMY UNIT DESTROYED
"
1710 PAUSE 50
1720 GOTO 80
```

```
2000 REM ******INITIALISE****
2003 LET A$="▛▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▜"
2006 LET A$=A$+"▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ ▌ "
2010 LET B$=""
2020 FOR X=1 TO 10
2030 LET B$=B$+A$
2040 NEXT X
2050 LET B$=B$+"▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀▀"
2055 RAND
2190 LET E$(1)="MV"
2192 LET E$(2)="RL"
2194 LET E$(3)="RR"
2196 LET E$(4)="FR"
2198 LET E$(5)="FD"
2300 REM **4 ENEMY PROGRAMS**
2310 LET Q$="0401040301010403010
4010301040104030401040401"
2312 LET ENEMY=2
2320 GOSUB STORE
2330 LET Q$="0402050403040101010
1030403040504050403050405"
2340 LET ENEMY=3
2350 GOSUB STORE
2360 LET Q$="0502030102030301030
4050402040302020501010203"
2370 LET ENEMY=4
2380 GOSUB STORE
2390 LET Q$="0401010102010104010
2010104010201010101020101"
2400 LET ENEMY=5
2410 GOSUB STORE
2490 REM ********************
2500 REM PROGRAM PROVIDED FOR
PLAYER TO TEST AGAINST ENEMIES
2510 LET Q$="0504050401050401050
4010504040205040105040101"
2520 LET ENEMY=1
2530 GOSUB STORE
2540 RETURN
3000 REM *******STORE*******
3005 LET Y=1
3010 FOR X=1 TO 22
3015 LET Z$=Q$(Y TO Y+1)
3020 LET D(X,ENEMY)=VAL Z$
3025 LET Y=Y+2
3030 NEXT X
3040 RETURN
5000 REM *******CHOOSE 2 START
LOCATIONS*******************
5005 RAND
5010 LET XLOC=VAL X$
5020 LET YLOC=VAL Y$
5025 LET LOCDIR=INT (RND*4)+1
5027 LET DIR=LOCDIR
5030 LET XP=XLOC
5040 LET YP=YLOC
5050 GOSUB DISP
5100 LET XFOE=VAL X$
5105 IF XFOE=XLOC THEN GOTO 5100
5110 LET YFOE=VAL Y$
5115 IF YFOE=YLOC THEN GOTO 5110
5117 LET FOEDIR=INT (RND*4)+1
5119 LET DIR=FOEDIR
5120 LET XP=XFOE
5130 LET YP=YFOE
```

```
5140 GOSUB DISP
5200 RETURN
6000 REM *******DISP********
6005 PLOT XP,YP
6010 GOTO 6050+DIR*10
6050 PLOT XP,YP+1
6051 PLOT XP-1,YP
6052 PLOT XP-1,YP-1
6053 PLOT XP+1,YP
6054 PLOT XP+1,YP-1
6057 RETURN
6070 PLOT XP-1,YP
6071 PLOT XP,YP-1
6072 PLOT XP+1,YP-1
6073 PLOT XP,YP+1
6074 PLOT XP+1,YP+1
6077 RETURN
6080 PLOT XP,YP-1
6081 PLOT XP-1,YP
6082 PLOT XP+1,YP
6083 PLOT XP+1,YP+1
6084 PLOT XP-1,YP+1
6087 RETURN
6090 PLOT XP-1,YP-1
6091 PLOT XP,YP-1
6092 PLOT XP+1,YP
6093 PLOT XP-1,YP+1
6094 PLOT XP,YP+1
6097 RETURN
62000 REM *******WIPE********
62005 UNPLOT XP,YP
62010 GOTO 6250+DIR*10
62060 UNPLOT XP,YP+1
62061 UNPLOT XP-1,YP
62062 UNPLOT XP-1,YP-1
62063 UNPLOT XP+1,YP
62064 UNPLOT XP+1,YP-1
62067 RETURN
62070 UNPLOT XP-1,YP
62071 UNPLOT XP,YP-1
62072 UNPLOT XP+1,YP-1
62073 UNPLOT XP,YP+1
62074 UNPLOT XP+1,YP+1
62077 RETURN
62080 UNPLOT XP,YP-1
62081 UNPLOT XP-1,YP
62082 UNPLOT XP+1,YP
62083 UNPLOT XP+1,YP+1
62084 UNPLOT XP-1,YP+1
62087 RETURN
62090 UNPLOT XP-1,YP-1
62091 UNPLOT XP,YP-1
62092 UNPLOT XP+1,YP
62093 UNPLOT XP-1,YP+1
62094 UNPLOT XP,YP+1
62097 RETURN
70000 REM *******MENU********
7001 SLOW
7010 CLS
7015 LET T=29
7020 PRINT ,,TAB 12;"MENU",,,TAB
 0;"WRITE NEW PROGRAM";TAB T;"1"
,,TAB 0;"TEST CURRENT";TAB 0;"PR
OGRAM AGAINST",,
7030 PRINT TAB 0;"KILLER";TAB T;
"2",,TAB 0;"SEEKER";TAB T;"3",,T
AB 0;"RANDOM";TAB T;"4",,TAB 0;"
RUNNER";TAB T;"5"
```

```
7050 PRINT ,,"PLEASE GIVE A CHOI
CE","FROM 1 TO 5"
7100 LET X=CODE INKEY$
7110 GOTO 7100+20*(INKEY$<>"" AN
D X>28 AND X<34)
7120 PRINT "CHOICE: ";CHR$ X
7125 PAUSE 10
7130 RETURN
7131 REM WITH X=29-33
8000 REM **FIND NEXT COMMAND**
8020 LET X=D(PTR,PHASING)
8100 GOSUB (8150 AND X=1)+(8200
AND X=2)+(8250 AND X=3)+(8300 AN
D X=4)+(8350 AND X=5)
8110 IF PHASING=PLAYER THEN LET
PPTR=PPTR+1
8120 IF PHASING=ENEMY THEN LET E
PTR=EPTR+1
8122 IF PPTR=23 THEN LET PPTR=1
8124 IF EPTR=23 THEN LET EPTR=1
8130 RETURN
8150 REM *****MOVE COMMAND*****
8152 GOSUB WIPE
8155 LET XP=XP+((DIR=4 AND XP<58
)-(DIR=2 AND XP>2))*4
8158 LET YP=YP+((DIR=1 AND YP<41
)-(DIR=3 AND YP>5))*4
8160 IF XP=2 AND DIR=2 THEN LET
DIR=4
8162 IF XP=58 AND DIR=4 THEN LET
 DIR=2
8164 IF YP=5 AND DIR=3 THEN LET
DIR=1
8166 IF YP=41 AND DIR=1 THEN LET
 DIR=3
8170 GOSUB 8475
8175 GOSUB DISP
8180 IF PHASING=PLAYER THEN LET
XLOC=XP
8182 IF PHASING=ENEMY THEN LET X
FOE=XP
8185 IF PHASING=PLAYER THEN LET
YLOC=YP
8187 IF PHASING=ENEMY THEN LET Y
FOE=YP
8190 IF (XLOC=XFOE) AND (YLOC=YF
OE) THEN GOTO 8192
8191 RETURN
8192 IF PHASING=PLAYER THEN LET
WIPEOUT=2
8193 IF PHASING=ENEMY THEN LET W
IPEOUT=1
8196 RETURN
8200 REM *****ROTATE LEFT****
8202 GOSUB WIPE
8205 IF DIR=4 THEN LET DIR=0
8210 LET DIR=DIR+1
8212 GOSUB DISP
8215 GOSUB 8475
8220 RETURN
8250 REM ****ROTATE RIGHT*****
8252 GOSUB WIPE
8255 IF DIR=1 THEN LET DIR=5
8260 LET DIR=DIR-1
8265 GOSUB DISP
8280 GOSUB 8475
8290 RETURN
```

```
8300 REM *******FIRE**********
8305 LET FX=XP
8308 LET FY=YP
8315 GOSUB 8450
8323 FOR X=1 TO 5
8326 LET FX=FX+((DIR=4)-(DIR=2))
*4
8329 LET FY=FY+((DIR=1)-(DIR=3))
*4
8332 IF FX>58 OR FX<2 OR FY<5 OR
 FY>41 THEN RETURN
8333 PLOT FX,FY
8334 UNPLOT FX,FY
8335 IF (TX=FX AND TY=FY) THEN G
OTO 8340
8338 NEXT X
8339 RETURN
8340 IF PHASING=PLAYER THEN LET
FDEF=FDEF-1
8342 IF PHASING=ENEMY THEN LET P
DEF=PDEF-1
8344 LET WIPEOUT=WIPEOUT+((FDEF=
0)*2)+(PDEF=0)
8346 RETURN
8350 REM ******FIND***********
8352 GOSUB WIPE
8355 GOSUB 8450
8360 IF XP<TX THEN LET DIR=4
8365 IF XP>TX THEN LET DIR=2
8370 IF YP<TY THEN LET DIR=1
8375 IF YP>TY THEN LET DIR=3
8380 GOSUB 8475
8385 GOSUB DISP
8390 RETURN
8400 REM ********************
8450 IF PHASING=PLAYER THEN LET
TX=XFOE
8455 IF PHASING=ENEMY THEN LET T
X=XLOC
8460 IF PHASING=PLAYER THEN LET
TY=YFOE
8465 IF PHASING=ENEMY THEN LET T
Y=YLOC
8470 RETURN
8472 REM ********************
8475 IF PHASING=PLAYER THEN LET
LOCDIR=DIR
8480 IF PHASING=ENEMY THEN LET F
OEDIR=DIR
8485 RETURN
8500 REM **COLLECT NEW PROGRAM*
8510 CLS
8520 FOR X=0 TO 21
8530 PRINT "COMMAND  ";X;":  "
8540 NEXT X
8600 PRINT AT 0,15;
8610 FOR X=1 TO 22
8620 INPUT S$
8630 GOSUB 8700
8640 IF Y=6 THEN GOTO 8800
8650 LET O(X,1)=Y
8652 PRINT TAB 25;S$
8655 NEXT X
8656 LET ENEMY=2
8658 LET PHASING=ENEMY
8659 GOTO 208
8660 REM ********************
8700 FOR Y=1 TO 5
```

```
8710 IF S$=E$(Y) THEN RETURN
8720 NEXT Y
8730 RETURN
8750 REM ************************
8800 PRINT AT 0,16;"INVALID";
8810 PAUSE 30
8820 PRINT AT 0,16;"       ";
8825 PRINT AT X-1,12,
8830 GOTO 8620
9000 REM ************************
9990 SAVE "ARENA"
9995 GOTO 80
```

The only criticism I have seen of the original *Robotwar* is that one of its built-in Robot programs has a disappointingly simple but optimum strategy, so that it can't be beaten. This seems to have something to do with the set-up, which is non-random; the optimum program "cheats" by relying on its "knowledge" of likely initial dispositions and catches the player's Robot before it has had time to manoeuvre. This approach is not possible in *Arena*, but there may indeed be a simple optimum strategy; however, it hasn't occurred to me yet.

There follow some notes on the way that the *Arena* program, listed in full in Listing 5.1, is structured, with explanations of how the various subroutines work. After that, we will consider room for possible improvement of the program and look at some of the conclusions that can be drawn from it.

In addition, here is some explanation of the functions of individual groups of lines in the program:

400-430: this is an example of the use of AND as a "flip-flop" to load one of two values into a local variable such as DIR or PTR. Once this has been done, such variables can be used in common subroutines which implement the Turns of either player. Turns alternate, so each pair of values is loaded into the appropriate local variable alternately.

2190-2198: the numeric equivalent of the "MV" command, for example, is 1. There's no need to store the value 1 anywhere, as the program notes that it has found MV in element 1 of the E$ array when it is searching for matches to command-codes input by the player.

2300-2510: the values held in Q$ are the numeric equivalents of the five command-codes, and the commands shown in the string will be executed in sequence. As two characters are used to hold each numeric value, this section of the coding could be used to hold a much wider range of numeric values to cater for an expanded version of the game which included additional commands.

3000-3040: the numeric values are loaded into the D array which the program will refer to when finding the next instruction to execute.

8155-8158: this is the awkward problem of altering the XP and YP values so that they reflect the unit's new position in the playing area. The unit is depicted as six PLOT points and the value for XP and YP will be the PLOT co-ordinates of the central point of the six (the routine at 6000-6097 works out where to put the other five, depending on the unit's facing). XP is either incremented or decremented depending on the unit's facing, and then multiplied by four which represents a move to an adjacent square on the grid. AND is used to decide whether to increment or decrement, and the same technique is used for the YP value. The lines also incorporate a check that the unit has not already reached the edge of the playing area, and all this can be contained in two lines. However, the routine to reverse the facing of the unit is contained separately, in the next four lines.

8326-8329: a similar technique is used to decide where the PLOT point should appear to represent the cannon firing, and again this is followed by a check for the edge of the playing area.

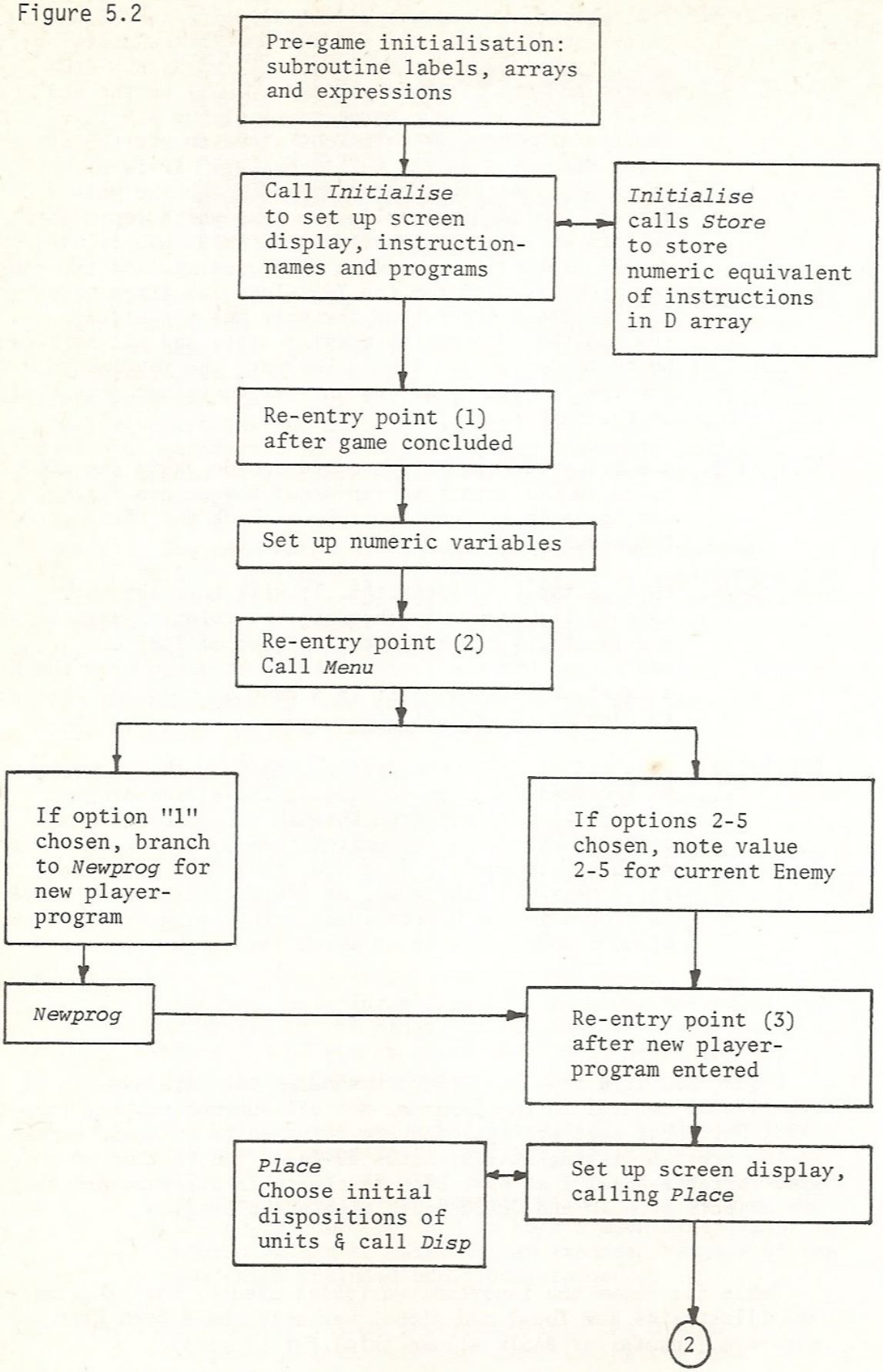
8360-8375: this is the Find algorithm. It will turn the unit towards its Enemy, by checking the relative values for the locations of the units in terms of PLOT co-ordinates, but the Y axis takes precedence over the X axis if, for example, it is a case of choosing whether to face South or West.

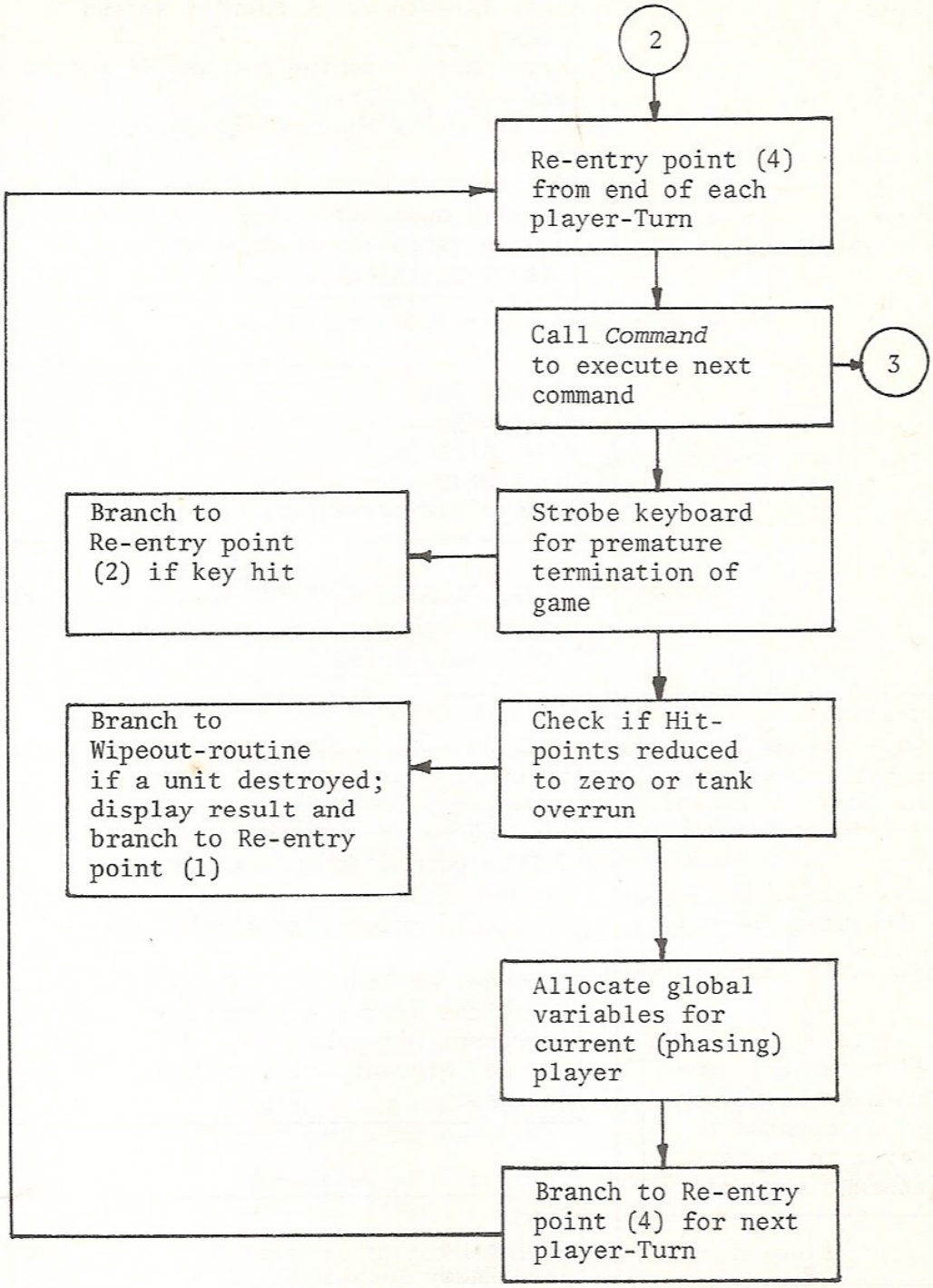
8700-8730: when collecting a new instruction from the player, to be included in a new version of the Player-program, this routine will search through the E$ array which contains the five instruction-codes, looking for a match. If it doesn't find one, line 8640 gives an error message. Otherwise, an equivalent numeric value is stored in the D array, the value being simply the element address in E$ at which the match was found.

Figure 5.2 is a type of flowchart showing the main aspects of the flow of control in the program. Not all subroutines are itemised, but those that are included are referred to by their names in the program listing itself; lines 30-48 of the Listing show some variable-names that have been equated with line-numbers that are objects of GOTO and GOSUB calls within the program.

Table 5.1 shows the important variables used by the program, and illustrates how local and global variables have been kept separate (insofar as BASIC allows this) for clarity.

Figure 5.2


Pre-game initialisation: subroutine labels, arrays and expressions
Call *Initialise* to set up screen display, instruction-names and programs
Initialise calls *Store* to store numeric equivalent of instructions in D array
Re-entry point (1) after game concluded
Set up numeric variables
Re-entry point (2) Call *Menu*
If option "1" chosen, branch to *Newprog* for new player-program
If options 2-5 chosen, note value 2-5 for current Enemy
Newprog
Re-entry point (3) after new player-program entered
Place Choose initial dispositions of units & call *Disp*
Set up screen display, calling *Place*
2


2
Re-entry point (4) from end of each player-Turn
Call *Command* to execute next command
3
Strobe keyboard for premature termination of game
Branch to Re-entry point (2) if key hit
Check if Hit-points reduced to zero or tank overrun
Branch to Wipeout-routine if a unit destroyed; display result and branch to Re-entry point (1)
Allocate global variables for current (phasing) player
Branch to Re-entry point (4) for next player-Turn

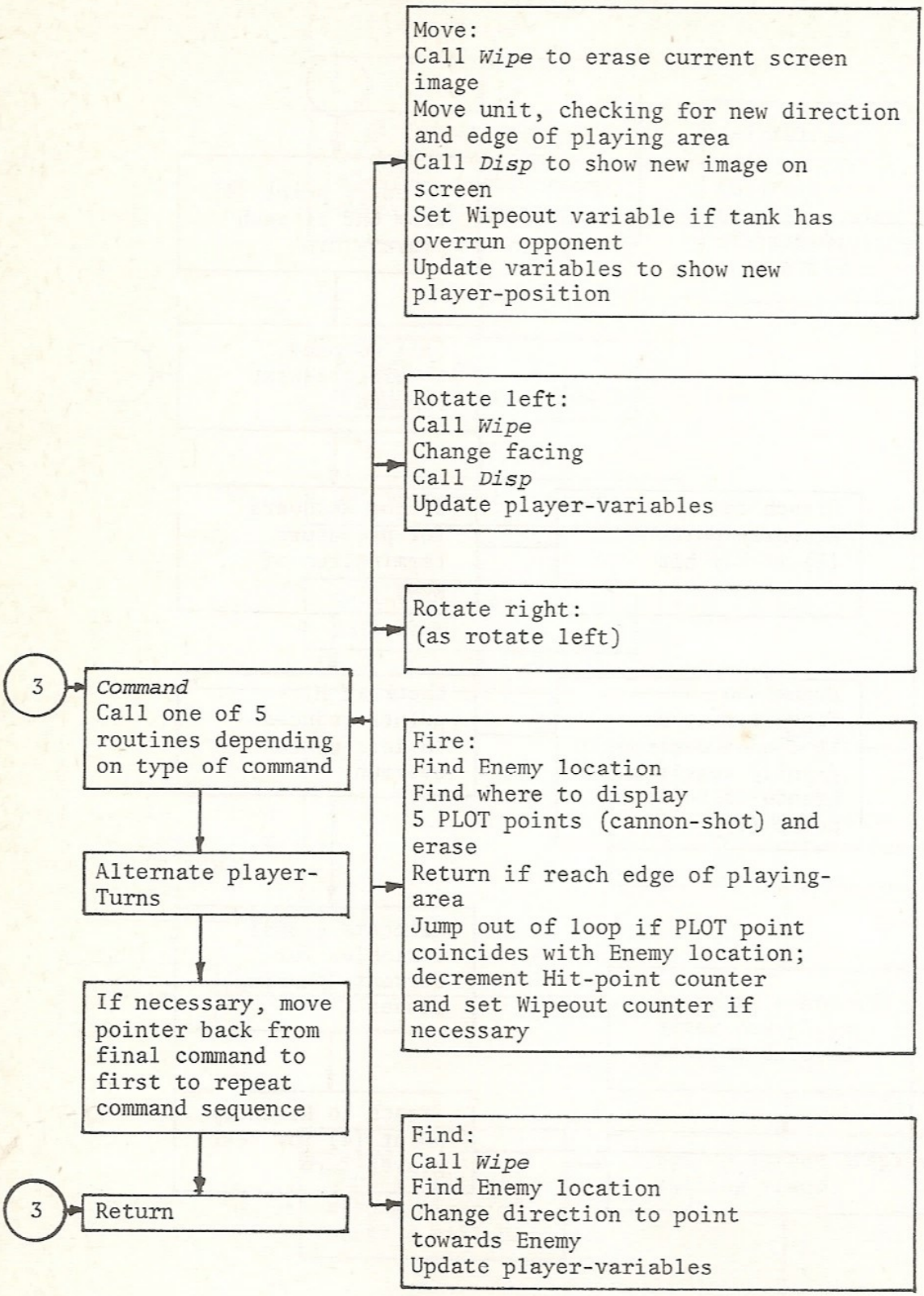

Move:
Call Wipe to erase current screen image
Move unit, checking for new direction and edge of playing area
Call Disp to show new image on screen
Set Wipeout variable if tank has overrun opponent
Update variables to show new player-position
Rotate left:
Call Wipe
Change facing
Call Disp
Update player-variables
Rotate right:
(as rotate left)
3
Command
Call one of 5 routines depending on type of command
Fire:
Find Enemy location
Find where to display
5 PLOT points (cannon-shot) and erase
Return if reach edge of playing-area
Jump out of loop if PLOT point coincides with Enemy location; decrement Hit-point counter and set Wipeout counter if necessary
Alternate player-Turns
If necessary, move pointer back from final command to first to repeat command sequence
Find:
Call Wipe
Find Enemy location
Change direction to point towards Enemy
Update player-variables
3
Return

Table 5.1

Table 5.1 ARENA significant variables

Arrays	
D (22,5)	holds 22 commands comprising "program", as numbers 1 to 5. Player-program in row 1, Enemy programs in rows 2-5.
E$ (5, 2)	holds two-letter commands such as MV.
String variables	
X$, Y$	hold string-equivalents of expressions (lines 70-72) for selecting random starting locations for the robot tanks at the start of the game.
Global variables	
XLOC, YLOC	Player's current location on screen.
XFOE, YFOE	Enemy's current location on screen.
PPTR, EPTR	pointer to next command to execute, for Player's unit and Enemy unit respectively.
LOCDIR, FOEDIR	direction of facing for Player's unit and Enemy unit respectively.
PLAYER, ENEMY	PLAYER=1 and ENEMY=2-5; these values are loaded into PHASING to show whose Turn it is.
FDEF, PDEF	Hit-point allocation for Enemy and Player units respectively; decremented as Hits received.
WIPEOUT	initialised as zero. 1=Player's unit destroyed, 2=Enemy unit destroyed.
Local variables	
XP, YP	current location on screen of Phasing unit, used by common subroutines.
DIR	direction of facing for Phasing unit. 1=North, 2=West, 3=South, 4=East.
PTR	pointer to next command for Phasing unit.
PHASING	value 1-4 representing Phasing player.

CONCLUSIONS

First of all, the *Arena* program is not being offered as the best program that can be written of its general type, and the reader may well be able to improve on it. It would become much more complex if two or three more commands were added to the "instruction set" that the robot tanks understand. A range-finding mechanism would be particularly useful, saving a great deal of wasted ammunition. The Find instruction would be greatly improved by a routine which allowed a tank to get at an opponent which was diagonally removed from it, and so on. The first thing that becomes clear when one tries to work out such improvements is that some kind of conditional instruction is required, along the lines of IF RANGE LESS THAN 6 THEN FIRE. This is hardly surprising; conditional instructions are one of the five or six essential components of any computer language, and the five-keyword language contained in *Arena* is just that - a language for controlling the actions of mechanical devices. It is a metalanguage in the sense that it is implemented through a language (BASIC) which is used to control the actions of the computer, but the user of the ZX81 is already familiar with this idea, since the BASIC language itself is in turn implemented through the use of packets of Z80 instructions contained in the ROM.

Writing your own computer language sounds like a daunting task, but, as it turns out, a simple language of this kind can be implemented in BASIC in 10K, leaving enough room in memory to add perhaps as many as seven or eight further commands. The language would then approach the complexity of the simplest computer languages that are in everyday use, such as Tiny BASIC. It should be borne in mind that the game-mechanics would become correspondingly intricate and it would take a long time to evaluate a new Player-program by running it against all the opponents; as the program stands, this is already time-consuming.

On the subject of time, the program runs rather slowly, but this is not so much because of any inherent complexity. It's because it was written for clarity and contains a number of features which make the program easier to understand, but result in slow execution. Primarily, the long routine-names hold up program flow considerably, and could be replaced by one-letter names. It would also pay to replace numeric literals like 58 and 41 with numeric variables, and all variable-names should be one or (if necessary) two letters in length. Various other speeding-up techniques, such as putting the subroutines at the beginning of the program, are discussed in Chapter 1. The comments about tradeoff apply, of course, and once the program had been altered in this way it would be a bit impenetrable.

Does *Arena* match the definition, given earlier, of a good game? At any rate it has some of the necessary qualities: the rules are extremely simple, the playing-area and pieces are satisfyingly abstract, and, although all the facts about the mechanics are known to the player, he will have considerable trouble in guessing in advance of a trial-run whether his latest program will beat the

opposition. As a matter of fact, it's comparatively easy to work out the result of a given game once you know the initial disposition and the "listing" of the two programs. It's quite a different matter to deduce in the abstract which program will win most often against a specified opponent, given random setups.

Arena is, therefore, offered as a simulation of machine intelligence which does not require a disproportionate amount of effort on the part of the programmer. It may be objected that, like the Draughts program, it is not intelligent because we have injected our own intelligence into it, in the form of strategically optimised programs. This is true, but think how easy it would be to make the program play one opponent against another, independent of participation by the player. For example, Killer can be matched against Runner by copying the contents of row 2 of the D array into row 1, and selecting option 5 on the menu. A natural extension of this is to make the *Arena* program generate its own robot-programs, by a methodical iteration of all the possible permutations of instruction-sequences. Probably it ought to test out each new program at least five times, to make sure that it was a winner in all circumstances. Then it would match it against the next automatically generated permutation, and so on. Admittedly this would take a long time, but there would be no need to stand and watch; one would go away for a week or two and leave the program running. At the end of the process, the computer would have done something we can't do: figure out optimum strategy for *Arena*. What we can do, and the computer can't, is write *Arena* in the first place.

But perhaps the project of writing a game such as *Arena*, which uses the concept of a metalanguage, gives us some insight into the way in which computers will in due course be able to manipulate symbols more effectively and more independently than they do at the moment. Probably such a machine will use a hierarchy of languages, from very primitive ones at the bottom of a stratified layer of available languages, to more abstract and powerful ones at the top. This is thought to be the way in which our own mental processes work; the fact is that we have to discover a great deal more about our own thought-processes before further progress can be made in this area. Games like *Arena* may be able to provoke us into some insights into the way our own minds work, and what the definition of "intelligence" really is.

Chapter 6 Artificial Intelligence: Language

> "Writing a program which can fully handle the top five words of English - 'the', 'of', 'and', 'a' and 'to' - would be equivalent to solving the entire problem of AI, and hence tantamount to knowing what intelligence and consciousness are."
>
> - Douglas R. Hofstadter,
> *Gödel, Escher, Bach: an eternal golden braid*
> (Basic Books Inc., 1979)

Hofstadter is not a pessimist on the subject of progress in Artificial Intelligence, but he is aware of the difficulties involved. This quotation is not meant to intimidate the reader, just to point out that this area of computing applications is a big problem and only a start has been made so far. So far as the ZX81 is concerned, there's a severe limit to what can be done on such a machine and we will look in this chapter at what is feasible.

Hofstadter also realises that AI research nowadays really means language research. There are a number of reasons for this, but it is essentially a question of our own limitations rather than the computer's. If we want to start asking really interesting questions, and expect to get answers that are not only interesting but comprehensible to us, we are going to have to use our own, natural languages to put the questions, and whoever or whatever gives the replies must do the same. We can invent any number of artificial languages or proposition-generating systems, and these can play an intermediate part in such a dialogue, but our definition of what intelligent behaviour means is inseparable from our views on the correct way to use natural language. Hence, a great deal of work has been done to make computers simulate natural discourse, the most advanced effort probably being the program SHRDLU; this allows a computer to give idiomatic and intelligent replies to a series of commands and questions concerning the movement of blocks, boxes and pyramids within a strictly defined universe.

However, the average computer enthusiast is more likely to have encountered other examples of programs which appear to understand and speak English. Programs of the Eliza, Doctor or Parry type are quite widespread and allow the user to hold quite an interesting conversation with a computer-generated adviser or Counsellor, along the following lines (human responses preceded by a dash):

YOU SAY YOUR RELATIONSHIP WITH YOUR BROTHER IS NOT AS GOOD AS IT WAS

-YES, HE HAS BEEN BEHAVING VERY ERRATICALLY

HOW LONG HAS THIS BEEN GOING ON?

-OH, SINCE OUR PARENTS WERE SEPARATED

DO YOU THINK HIS ERRATIC BEHAVIOUR IS LINKED TO THE SEPARATION OF YOUR PARENTS?

-I'VE BEEN THINKING ABOUT THAT

IF YOUR RELATIONSHIP WITH YOUR BROTHER WAS BETTER, YOU COULD DISCUSS YOUR THOUGHTS WITH HIM

This sounds very impressive, but there is not even a simulation of intelligence here. A databank of stock phrases is held by the program, and built up into sentences by interleaving them with fragments of the human's replies whenever these replies contain keywords which the program is looking out for. Due attention is, of course, paid to grammatical rules governing such things as construction of plurals; it helps considerably that most people, when in discussion with a Doctor or similar figure, expect to have the conversation constantly turned back on their own emotions and do not actually want to know anything about the Doctor's feelings. However, once one knows how the trick is done, the program can easily be led hopelessly astray; it has no defense against such tactics, since it carries out no analysis at all on most of the phrases which it receives and then spits back:

YOU SAY YOUR RELATIONSHIP WITH YOUR BROTHER IS NOT AS GOOD AS IT WAS

-YES, I THINK IT'S BECAUSE I CASTRATED HIM

HOW LONG AGO DID THIS HAPPEN?

-OH, AT THE SAME TIME THAT IT ELECTROCUTED MY PARENTS

DO YOU THINK YOUR POOR RELATIONS WITH YOUR BROTHER ARE LINKED TO THE ELECTROCUTION OF YOUR PARENTS?

-NO, HE HELPED ME DO IT

PERHAPS THIS PROVIDES THE COMMON GROUND ON WHICH YOU CAN IMPROVE YOUR RELATIONSHIP WITH YOUR BROTHER

Writing Eliza-type programs can be a lot of fun and is not such a difficult task as might be expected, but it doesn't really have anything to do with AI. A more interesting challenge is posed by the problem of writing a language-input analyser for an Adventure-type game. One of the more sophisticated games of this genre that is available on microcomputers, *Zork,* was originally conceived as an AI research project, centred around the task of giving intelligent responses to an individual (the player) who had been transposed to an imaginary environment and wanted to act within it.

A certain degree of protection is provided to such a program because its universe is strictly defined in advance and it only needs to have competence within that universe; it is entitled to dismiss as irrelevant any attempt to lead it into discussion of matters outside this domain.

Even so, programming a sentence-parser for *Zork* was a major effort and brought home to the programmers the extreme subtlety of everyday English. To see what sort of problems have to be overcome, let's begin by studying the simplest possible type of Adventure and working up to the more advanced examples of the genre.

The easiest arrangement from the programmer's point of view is to use single-key entry of a limited number of one-word commands. Probably the bare minimum would be:

```
F       FIGHT
R       RUN (AWAY)
L       (GO) LEFT
U       (GO) UP (and so on for down, back etc.)
G       GET
D       DROP
```

Many of the games available for the ZX81 are indeed of this type, and share some of the characteristics of the typical Adventure game. The player controls a single puppet figure which can move around in an environment divided into a number of locations; the environment is limited and its exact nature is fixed either at the beginning of each game, or by the program itself (that is, there is no random element involved in the arrangement of locations and the objects within them). The player has to discover the nature of the environment during the course of the game, by moving around in it, and can interact with it to the extent of obtaining portable objects found in the locations, transferring them from one location to another, and leaving them there.

Using single-key entry makes it very easy to write the input routine that the program uses to find out what the player wants to do, at the beginning of each cycle of the program. Little more than a single INKEY$ statement is required, and the choice of options open to the player is so limited that very little processing is needed to work out the correct response to be made by the program to the player's action. The problem is, of course, that the program is handicapped by the simplicity of its own vocabulary, that is, the list of terms that it understands when they are input by the player. Typically, the player will be confronted by a monster of some kind when he enters a particular location as a result of using the movement commands. He is supposed to exercise his skill and judgement to decide whether to fight or run away, basing his decision on the program's description of the monster. To add complexity to the game, the player's chances of defeating the monster will vary depending on the portable objects that he is holding, for example, magic swords and so on.

Supposing that the player is holding enough aids to be able to vanquish a given monster, the programmer has to decide how this is to be represented as a screen display. The problem that arises here is that, whatever happens, the player will be reduced to a spectator. Ideally, a series of rounds of combat would ensue in which the player had to struggle mightily with the monster before eventually overcoming it. The vocabulary does not, however, permit the player to struggle mightily, feebly or in any other way; he just fights. The programmer may decide to spin out the combat, either by inserting a graphic display or by printing a lengthy narrative describing the fight; either way, the player can ignore this section of the game as it can have no influence on his subsequent input decisions. Alternatively, the program may just print out MONSTER DIES and allow the player to get on quickly to the next location.

Programs of this type often include a large number of locations, sometimes over 200, and many different monsters, in compensation for the lack of a proper input-analysis routine. However, this can't really affect the complexity level of the game and make it more interesting to play. The only yardstick that is relevant when evaluating an Adventure is the size of its vocabulary, since this is what governs the range of actions the player can take and, in turn, the range of situations that can be built into the game as puzzles to be solved. To take an example from one of the Scott Adams Adventures, you may decide that bees should be put in a bottle in order to be transported from one location to another. This feature can't be included until you provide the player with some way of issuing a command equivalent to:

```
TAKE THE BEES AND THE BOTTLE, AND PUT THEM IN IT
```

This seems to plunge us into a much trickier problem than the programmer of the single-key input game had to face, involving full-scale sentence analysis. We know from earlier chapters in this book how to use a routine to collect multi-character input from the user, and it is not too difficult to break this sentence down into its component parts by regarding it as a string and using the TO slicer, but this is only the beginning of the task. To get an idea of the problem, imagine how many different versions of this example sentence you could write down which would all convey the same meaning to a human listener. The program will have to be able to understand all of them; on top of that, it will have to understand (but to reject as invalid) the set of all sentences which are meaningful but inappropriate under the current circumstances (the player is already holding the bees; the player is not holding the bees but is already holding the bottle; the player is holding neither but the bottle has been left in an adjacent location; the bees and the bottle are both here but the bottle broke when it was dropped three cycles ago; etc; etc.).

The solution most commonly adopted, which is a compromise, is to allow only two-word sentences and allow for them to be strung together where appropriate. The example sentence given above could

be implemented as follows (player input preceded by a dash):

```
-TAKE BEES
OK
-TAKE BOTTLE
OK
-STORE BEES
IN WHAT?
-IN BOTTLE
OK
```

The program must still make all the checks for valid input that it would have to make if the input were a single sentence, but in practice the coding will be much less complex and occupy less memory. For instance, action taken by the program will always be the same for every example of player-input in which the first word is TAKE: find the record of the player's current location, find the record of the current location of the object named in the second word of the sentence, and print the message I DON'T SEE THAT HERE if the two do not match.

I would not describe such a program as a parser, simply because it does not have to make any decisions about parts of speech. The first input word is found in a table of the words understood by the program, and the usual approach when writing such programs is to store a numeric value next to each word, which can be collected by the look-up routine and saved in a variable for subsequent processing. Then the second word is found in a second table of nouns that the program understands, and a second numeric value is derived. If either of these searches fails, the input is invalid and the user has to find some other combination of words. The program never has to consider a word in the context of another word in order to decide its meaning or grammatical correctness.

That is not to say that the programmer's problems are over; the logic of the program still has to be able to decide whether the particular two-word combination is sensible under the current circumstances and what action it should take as a result. Before we move on to more complex sentence analysis, we had better be sure that the ZX81 will be able to handle two-word input in an Adventure of reasonable sophistication, and a study of the techniques necessary to do this will underline the amazing multiplicity of inputs that can be generated by such an apparently restricted input-format.

In Chapter 1 a short routine was given using an array V$ to allow a branch to one of sixteen lines, by exploiting the expression-evaluation facility which is one of the ZX81's few real advantages over versions of BASIC on other computers. Now we have found an application which matches this facility; we can handle the permutation of meanings arising from all possible combinations of two words in a sentence in a very lucid fashion, by cross-

referencing the first word with the second in a two-dimensional array, and extracting an array element which will tell the program what to do next.

In order to illustrate the method, I've made up a vocabulary for an admittedly rather unimaginative Adventure, and the details are shown in Figure 6.1. There are only five verbs and eight nouns, but the same principle can be used for larger vocabularies without slowing down program execution, which is reasonably fast for a BASIC Adventure. As a matter of fact the eight words which can be the second word in the input sentence are not all nouns, and there is a slight element of grammatical subtlety in allowing the use of the word "down".

Figure 6.1

```
ARRAY V$ (5,8)


        NOUNS
        1=N**  2=S**  3=E**  4=W**  5-WAN(d)   6=BOX   7=DOW(n)   8=HOL(e)
VERBS
1=GO*     A      A      A      A       B         B       A          A

2=GET     C      C      C      C       D         D       A          C

3=DRO     C      C      C      C       E         E       C          C

4=WAV     C      C      C      C       F         H       C          C

5=LOO     G      G      G      G       I         J       G          K


 A= (1) check he can go that way (2) "OK"

 B= "DO YOU WANT ME TO TAKE AN OBJECT?"

 C= "I DON'T UNDERSTAND"

 D= (1) check object in same room (2) "I'VE TAKEN IT NOW"

 E= (1) check he is holding that object (2) "I'VE DROPPED IT NOW"

 F= (1) check he is holding the wand (2) "I'M ENVELOPED IN A
    MISTY GLOW"

 G= "OK"

 H= (1) check he is holding that object (2) "OK"

 I= (1) check he is holding the wand or it is in the same room
    (2) "IT SAYS 'WAVE ME'"

 J= (1) check he is holding that object or it is in the same
    room (2) "OK"

 K= (1) check he is in the same room as the hole (2) "OK"
```

The array can be set up by the use of a number of lines such as:

```
50      LET V$(1)="AAAABBAA"
```

which can be deleted later in program development if memory is tight, once they have been preserved in the variable table. The letters in the array are to be equated with numeric variables which hold values for line-numbers, so one also needs a series of lines such as:

```
100     LET A=2000
110     LET B=2200
```

and so on, and these can also be deleted in due course. If this is done, you may well need a little subroutine for use in debugging, to remind you of the current values stored in the vocabulary array; Listing 6.1 is an example of such an aid.

Listing 6.1

```
9000 FOR Z=1 TO 5
9010  FOR Y=1 TO 8
9020 LPRINT V$(Z,Y);"=";VAL V$(Z
,Y);" ";
9030 NEXT Y
9040 LPRINT
9050 NEXT Z
```

The five verbs in the vocabulary are GO, GET, DROP, WAVE and LOOK. The way they are shown in Figure 6.1 is the way they would most likely be stored, perhaps in another string array, with characters after the first three chopped off and a dummy character such as an asterisk used to pad out two-letter words. An INKEY$ subroutine is made up, similar to those shown in earlier Listings in this book, which will accept two words, taking the input of (say) a full stop to signify the end of one word and the start of the next. It then discards letters after the first three of each word, and adds asterisks as necessary, after which the program searches through the permitted words stored in its dictionary-array to find matches, returning with element-addresses from one to five (verbs) and one to eight (nouns).

The nouns are also shown in Figure 6.1, and you can see that only the initial letters of the four points of the compass have been used to denote these four words. The effect of this, when used in conjunction with the INKEY$ routine described above, is that the player can go north just by typing GO N, as the program will fill in the missing characters with asterisks; players like to have such shorthand methods of entering commands.

Once the program has found the two element-addresses, which might be held as the variables FIRST and SECOND, the programmer

can simply code:

```
1000   GOTO VAL V$(FIRST,SECOND)
```

The effect of this is that if, for example, the input is DROP BOX, V$(3,6) will return "E". This will be looked up in the variable table by the interpreter which will evaluate E as the line number chosen by the programmer when setting up initial values; thus, if E=3000, the program will GOTO 3000 where the programmer will have put a specific routine. In the example given, the routine must check that the specified object is being held, and if so print a message such as "I've dropped it"; if it isn't being held, it can display a message such as "I'm not holding it to start with".

Other actions to be taken by the program are listed in the diagram, and are meant to cover all conceivable circumstances that can arise from the combinations of first and second words. Of course, there may be other checks to be made; perhaps it's dangerous to go down the hole if one is not carrying the wand, and so on. Allowing GET DOWN but not GET HOLE is a nice touch because it gives the impression that the program has some grasp of idiomatic usage, but at the same time will reject anything that is clearly not normal English.

Normally an Adventure will have a larger vocabulary than this, and the programmer may well find himself running out of single-letter variables. Probably a larger array of three dimensions (the third holding two characters) will have to be used, and then single-letter numeric variables can be reserved for FOR/NEXT loops and other purposes. Even so, a vocabulary of 25 verbs and 60 nouns will only occupy 3010 bytes in the variable table, not including the space taken up by the list of truncated three-character words themselves. This may sound like a lot, but it replaces a great many tedious IF-THENs which would otherwise have to be coded. Generally, the memory consumption of an Adventure goes mainly on the logic that decides what to do as the result of a given input, and of course on the messages. Storing the map of the environment and keeping track of the current status of objects is less of a problem.

Even larger vocabularies could be used, but in practice it would be very surprising if the programmer could make use of the extra words without running out of 16K memory. Remember, 25 words and 60 nouns means 1500 possible permutations. Admittedly many of them don't make sense, and can be handled by the general-purpose "dustbin" routine used for option C, which just rejects the input as meaningless. On the other hand, many of those that do make sense require further processing in several stages. There is an element of duplication here; consider options E, F, H, I and J, which all require a check to be made that an object is being held. In programming terms, the way to keep track of this sort of thing is to have a variable for each object, the value of which denotes its location, and to include not only all the rooms of the envir-

onment but also the player's "pocket" as locations. Then a routine can be written which checks IF OBJECTLOCATION=POCKET to see if the player has the object.

It might therefore seem that such a routine, option L, should be created and that control should be sent to it first, instead of to E, F, H, I and J. The trouble is that the circumstances are slightly different in each case. With option E, the command is accepted if the player is holding *any object that can be held*. With option F, on the other hand, the special message (which presumably will have deep significance within the context of the game) is printed only if *an object is being held which is the wand*. With option H, the criterion is as option E but the message is different; with option J, the only criterion is that the named object must *either* be in the room (on the floor) *or* be held.

There are, of course, various ways around this duplication. It will almost certainly pay to create a subroutine which can be called with GOSUB from various points in the program, and will check whether a specified object is being held. It's also possible to create secondary string arrays, also containing variables representing line numbers, which will allow these further permutations of possibilities to be tabulated. If the Adventure contains a large number of "room-specific" events, which occur only in certain locations, then a one-dimensional array of the rooms themselves may help in directing execution to the appropriate routine.

Several things are apparent from the above account. One is that writing Adventures is by no means as enjoyable as playing them. In the main, it's an extremely dull business involving the methodical itemisation of every situation that can occur during play, so that an appropriate message can be included to cover it. The programmer doesn't have to cover every specific contingency in this way; if a particular action is specious in the context of the plot of the Adventure, he can just let it fall through to the default "I don't understand" routine and make the player try again. However, the penalty he pays is that his program will appear more stupid as a result; it has been maintaining an illusion of comprehension of the player's input, and it can be quite disconcerting when the illusion is suddenly dropped in this way.

Other conclusions that can be drawn concern the inherent limitations of this approach to sentence analysis. A vocabulary of 25 verbs and 60 nouns produces permutations of meaning which can be sorted out in the array V$ using 3010 bytes of memory, as described earlier, but there will probably be branches to about 80 different routines to handle the results. 80 two-letter numeric variables require another 560 bytes in the variable table, and probably each routine will need an average of 100 bytes to implement the appropriate responses and updating, leaving rather a small amount of memory in a 16K system for messages, a map of the environment and all other subroutines. This is partly because ZX81 BASIC is rather profligate of memory. It would be possible to design a specialised Adventure-programming language which would optimise memory consumption, and indeed this is exactly what was done when the *Zork*

Adventure was reprogrammed to fit onto microcomputer-based systems. The real culprit, however, is the English language, which can generate enormous numbers of meanings, and shades of meaning, from a small dictionary of words. This will be brought home to us if we actually write an Adventure and ask people to test it out; many of them will, for example, type in the word TAKE instead of GET, and we will have to program in a wide range of synonyms to cover this which will cope with such variant input, although the complexity of the game itself will not be increased at all. As a matter of fact the Adventure-writing scheme proposed above is designed to handle synonyms rather easily; they have to be added to the dictionary of words understood by the program, in their truncated three-character form, but it is not necessary to create a new numeric value to correspond to the meaning of the synonym. If the numeric equivalent for GET is 20, for example, the same value can be stored for TAKE; this ensures that the program is directed to the same row of the V$ array whichever word is used.

Unfortunately, this destroys the capability of our Adventure to make fine discriminations among idiomatic nuances! We have just arranged for the program to understand GET DOWN as an indication that the player wants his puppet character to lower himself into a hole in the ground which is to be found in one of the locations in the game. Now we realise that the program will regard TAKE DOWN as an equivalent command, although it is meaningless in these (but not in other) circumstances. Of course, it is unlikely that the player will type in this combination of words since it would be regarded as meaningless by himself. If he does, though, our program will be "caught out" immediately. Its apparent command of the English language is not so fraudulent as that of the Eliza-type program, but it is a fake nevertheless.

GET, TAKE and DOWN look like pretty harmless words and one probably would not appreciate their subtlety under any other circumstances; writing an Adventure of the kind described in this chapter is the quickest way to an understanding of the problems facing any AI programmer who ventures into natural language. Remember, we are still at the level of simple verbs and adverbs; we have not yet tackled Hofstadter's dreadful prepositions such as OF and TO. The trouble we have encountered has arisen from the transition from one-word to two-word inputs. If we now look at the problems of "real" English sentences, difficulties multiply.

Zork allows sentences such as: BURN ALL THE BOOKS BUT THE BLACK ONE. This looks like an advance, but there are a number of restrictions. One is that no adverbs at all are permitted. This seems odd, as it should not be too difficult (given that one has got as far as this typical *Zork* sentence) to figure out how the sentence parser should cope with adverbs. The reason, as one of the authors of the program has explained, is that the game itself is too simple to justify their inclusion. Typically, the player may want to type in:

```
-ENTER THE ROOM STEALTHILY
```

when going into a previously unexplored location, as this would be

prudent (*Zork* has a number of monsters). The trouble is that whatever happens to the player when he enters the room is fixed in advance; it makes no difference whether he enters it stealthily, impetuously or backwards. He would be wasting keystrokes by typing in qualifications of this kind; the program, although extremely sophisticated, does not allow for subtle differences in behaviour on the part of the player-character. Even if it did, further revisions of the game-situation would be needed; there would have to be circumstances in which it made sense for the player to move stealthily, and others in which he might be expected to behave differently.

This illustrates the point that the complexity of an Adventure is determined by the complexity of the permitted player-input; the full range of inputs must be worked out in advance, and game-situations can then be devised to fit some of them, the rest being handled by default or "dustbin" routines. The result will be a complex but very fragile network; adding just one more word to the vocabulary can bring down the whole edifice. This can happen because the player will use the new word, in combination with the existing vocabulary, to create a whole new class of meanings, all of which are valid in normal English usage. The programmer must then re-examine all the existing permutations of meaning to study the effect of adding the new word to them, and create a new set of messages and program-actions which cope with the new meanings. This is, however, much easier in a two-word-sentence Adventure than in one which "understands" full sentences; the new word, if it is a verb, can only be used in combination with one of the existing nouns, so the number of cases of new meaning to examine is limited to the number of nouns in the vocabulary.

How does one analyse a sentence such as BURN ALL THE BOOKS BUT THE BLACK ONE? If we are using ZX81 BASIC, the best approach will be to break the sentence down into its component parts. Probably we would arrange for the INKEY$ routine to store the individual words of the sentence in a string array; if this is set up to be able to handle up to 25 words, this should impose no practical restriction on the player. Each word is then held in a different "row" of the array. The first step is to cut out definite and indefinite articles; the level of analysis we will be attempting will be so crude that the meaning we derive from the sentence will not be affected by this. Each word can then be looked up in a dictionary-table which must also specify the part of speech it belongs to. We will end up with the following information:

verb	qualifier	object,plural	qualifier	identifier	object
BURN	ALL	BOOKS	BUT	BLACK	ONE

It will be clear from this that the program's idea of English grammar is somewhat different from our own; it is a streamlined version which is all that the program needs for its analysis.

The next stage involves replacing ALL, BUT and ONE by more definite terms corresponding to data items held in the program's

database of objects and object locations. If there is more than one book, each will be identified to the program by a unique token; let us describe these tokens as BOOK1, BOOK2 and BOOK3. Each object will have its own list of characteristics, which will include its current location. Suppose that the program establishes that all the books are at the current player-location, and have the following additional characteristics:

```
BOOK1  RED
BOOK2  BLACK
BOOK3  GREEN, HEAVY
```

The next step is to replace certain terms in the input sentence with categories that the program can understand. The string array can be amended to look something like this:

```
BURN   BOOK1,BOOK2,BOOK3   BUT   BLACK   ONE
```

In practice, the programmer will not want to stay at the string-level for too long; he will find it easier to shift the words into a numeric array, using numeric equivalents for each word. First, though, the idioms in the original sentence have to be replaced by expressions that are easier to process. The rule with ONE is that it always refers to a physical object, and that this will be the last-quoted object in the case of a sentence containing references to more than one type of object. The rule with BUT is that it cancels out a previous reference to an object (the parser is not able to cope with the full range of meanings that the word BUT can have in normal English). Putting these facts together, we can amend the sentence to read:

```
BURN   BOOK1,BOOK3
```

This is a considerable advance over the original, but of course a number of stages of processing were needed to get to this point. The program must be able to recognise that BOOK1, BOOK2 and BOOK3 all belong to the category BOOK, and so it will not be enough to store each of these objects as a single identifying numeric value. More likely, the characteristic BOOK (or its numeric equivalent) will be stored along with other characteristics such as RED and HEAVY for BOOK1, BOOK2 and BOOK3. Then, when the program is contemplating the word BOOKS in the original input sentence, it can perceive that it is a plural and search through the list of characteristics to find all objects that have the characteristic BOOK. Having also found the objects that have the characteristic BLACK (only one of them in this case) it can subtract it from the list of books to be burned.

This account omits reference to a number of subroutines that will have to be added if the program is to maintain its apparent comprehension of English. For example, in the circumstances described, the program ought to reject BURN ALL THE BOOKS BUT THE BLACK ONES since there is only one black book. This means creating

a subroutine which checks that the player never tries to refer to a singular object in the plural, and vice versa. The subroutine would have to be called every time an input sentence needed to be analysed, although under normal conditions it would have no noticeable effect on play (except to slow it down).

After considering the depth of processing needed to cope with multi-word sentences, I think we will conclude that a 16K machine is not the right vehicle for such applications. Surprisingly, all the statements that we need in a programming language that can handle such a task are present in ZX81 BASIC; we might wish for some extra statements that would provide short cuts, such as a statement FIND that would search for a given string in an array and return its location, but these are not essential. What makes the proposition impractical is the exponential increase in the total number of meanings that have to be analysed, caused by the increase in the size of the vocabulary and the number of words in the input sentence. It is rather like the exponential increase in the number of moves that have to be considered by a look-ahead Chess program, but there are two important differences.

One difference is that, although the structure of a sentence-parsing program can certainly be flowcharted or diagrammatised, the resulting visual depiction of the workings of the program will be a great deal more untidy than a corresponding diagram of a Chess-playing program. The latter will operate in a methodical way; it will consider in turn all the possible moves it can make. For each one, it will consider all the responses that the opponent might make. For each response, it considers each valid second move that it might make, and so on for as far ahead as it is programmed to look. The sentence parser, on the other hand, must break down the original input and build it up again as a different structure, one which is more amenable to analysis. The structure that it builds up will vary widely depending on the type of input sentence, and will not be created according to the same rules for each sentence.

The other difference is that the sentence parser spends most of its time trying to decide whether it understands the input, or whether it should reject it as invalid. A Chess program will, of course, check for input that conflicts with the rules of Chess, but this will be a very small part of the complete program and the checking can be done in the abstract. The program needs to "know" the rules and needs to be aware of the board position, in the sense that the destination square given by a player for his piece must be one that this piece can reach from its current position. All this is, however, divorced from the in-depth analysis of the board position that constitutes the program's claim to intelligence.

By contrast, the parser will have to progress almost to the end of its analysis before it can decide whether the player's input is valid in the program's terms. Sentences which are perfectly acceptable in some circumstances must be rejected in others, because the player's environment is different. In order to perceive that the player's input is incompatible with his environment, however, the

program first has to decide the "meaning" of the input. The fact that the validity of a statement cannot in practice be decided without reference to its meaning (in other words, cannot be evaluated by a mechanical examination of the words in the sentence, without regard to the environment) is the major stumbling-block in the way of programmable linguistic analysis.

Despite all these cautions, the reader may still want to attempt the construction of a sentence-parser on the ZX81, even if it fills up all the memory by itself. It should be pointed out, though, that this is not necessarily the route that AI research is taking. The program SHRDLU, referred to at the beginning of the chapter, works in quite a different way to the *Zork*-type parser we have been describing. Instead of waiting until the input has been collected and then breaking the whole thing down into simpler structures, it starts work as soon as the first phrase comes in, simultaneously carrying out the activities of parsing, semantic analysis and deduction. This procedure of narrowing down the range of possible meanings of a sentence to one unique meaning corresponds much more closely to the way that human language understanding works, but it also demands much more powerful computers!

The two-word sentence analyser described in this chapter is on the other hand suited to the capabilities of the ZX81, and the reader who has the patience to work out a plot and then implement it in the manner described should be able to come up with a successful program. There is, of course, a different approach to language manipulation on a computer, which is to concentrate on the machine's ability to generate language rather than its powers of comprehension. Programs of this kind are a great deal easier to write, but perhaps have little to tell us about Artificial Intelligence. They rely on grammatical rules which are embedded in the program, and which are used to select words from a sectionalised vocabulary.

A modest example of such a program is shown in Listing 6.2, and it will generate profound utterances about existence for as long as you want to leave it running. It can, of course, be given a much larger vocabulary than the one included in the program, and this will avoid the problem of repetition of words from one sentence to the next. The first row of the array A$ should be filled with words like "some" and "every", spaced out to ten characters each by adding full stops as appropriate. Row 3 holds nouns, and the trick here is to use abstract nouns which will give an impression of profundity irrespective of the context in which they appear. As you can see, row 2 holds adjectives, but lines 130 and 400 give only a 50% chance that an adjective will be printed in the output sentence, just to add a touch of variety. Row 4 is used for any verb-auxiliary that will fit in the grammatical context, and the final row must hold transitive verbs. If you want more than six words in each section, change line 1000 so that the RND multiplier matches the number of words.

The subroutine at 1000 picks a word from the appropriate section and allocates it to B$. The subroutine at 7000 prints the word on

Listing 6.2

```
    2  SLOW
    4  GOSUB  5000
    6  RAND
    7  LET  Y$=""
   10  DIM  A$(5,60)
   20  LET  A$(1)="THE.......EACH..
...EVERY.....NO.........THIS....
..SOME.....
   30  LET  A$(2)="DIVINE....CRUEL.
.....DEAD.....ENDLESS...PERFECT.
..UNKNOWN...."
   40  LET  A$(3)="HOPE.......SPIRIT
....TRUTH.....DECEPTION.ANSWER..
..UNIVERSE.."
   50  LET  A$(4)="SHOULD....MUST..
.....CANNOT.....MAY.......WILL....
..WILL  NOT.."
   60  LET  A$(5)="LOVE.......NEED..
.....KNOW.....SEE.......DESTROY.
..NEGATE....."
   90  CLS
   95  FOR  R=1 TO 5
  100  LET  P=1
  110  GOSUB  1000
  115  LET  Z$=B$
  120  GOSUB  7000
  130  IF  (INT  (RND*2)) THEN GOTO
200
  140  LET  P=2
  150  GOSUB  1000
  155  LET  Y$=B$
  160  GOSUB  7000
  200  LET  P=3
  210  GOSUB  1000
  215  LET  X$=B$
  220  GOSUB  7000
  250  LET  P=4
  260  GOSUB  1000
  270  GOSUB  7000
  280  PRINT
  300  LET  P=5
  310  GOSUB  1000
  320  GOSUB  7000
  350  LET  P=1
  360  GOSUB  1000
  370  IF  B$=Z$ THEN  GOTO  350
  380  GOSUB  7000
  400  IF  (INT  (RND*2)) THEN GOTO
450
  410  LET  P=2
  420  GOSUB  1000
  430  IF  B$=Y$ THEN  GOTO  420
  440  GOSUB  7000
  450  LET  P=3
  460  GOSUB  1000
  470  IF  B$=X$ THEN  GOTO  460
  480  GOSUB  7000
  500  PRINT
  505  PRINT
  510  NEXT  R
  600  PRINT AT 18,0;"PRESS ANY KE
Y  FOR  FURTHER          REVELATIONS"
  610  GOTO 610+(INKEY$<>"")
  611  GOTO  90
```

```
  800 REM *****SUBROUTINES****
 1000 LET X=(INT (RND*6))*10+1
 1010 LET B$=A$(P,X TO X+9)
 1100 RETURN
 5000 CLS
 5010 PRINT AT 4,12;"THE";AT 7,9;
"SECRETS OF";AT 10,8;"THE UNIVER
SE"
 5020 PAUSE 80
 5100 RETURN
 7000 FOR C=1 TO 10
 7010 IF B$(C)="." THEN GOTO 7500
 7020 PRINT B$(C);
 7030 NEXT C
 7500 PRINT " ";
 7510 RETURN
```

```
SOME DECEPTION MUST
DESTROY THIS UNIVERSE

EACH TRUTH SHOULD
KNOW EVERY DEAD HOPE

EACH SPIRIT WILL NOT
SEE THIS DIVINE TRUTH

THE UNIVERSE MUST
DESTROY EVERY TRUTH

EACH DEAD DECEPTION WILL
NEED EVERY TRUTH



PRESS ANY KEY FOR FURTHER
REVELATIONS
```

the screen, followed by a space. The first three words of each sentence are saved in Z$, Y$ and X$ so that the program can check that it is not repeating them when it makes up the final three words.

When adding vocabulary, it might be a good idea to stress technical terms drawn from Existentialism and German philosophy; anything too literal might give the game away. If the vocabulary is large enough, and the program is left running long enough, I suppose it is bound to generate some quite perceptive insights. Does this mean that the ZX81 is intelligent? The answer must be no; not only is the ZX81 unable to produce a justification for any of the pronouncements it makes, but it has no idea which of them are true and which are nonsense. Even someone who is quite naive about the power of computers is unlikely to believe that the machine is "intelligent" just because it is capable of generating an infinite number of sentences. It would be different if it could analyse its own output and only print sentences that offered a genuine advance in human understanding. That day, unfortunately, is still a long way off.

Chapter 7 Using assembly language on the ZX81

One of the themes of this book has been that the programmer who wants to do advanced work on the ZX81 will sooner or later be frustrated by the limitations of ZX81 BASIC, and will need to use machine-code programs or routines in order to achieve particular effects or simply to speed things up.

Using machine code on the ZX81 presents special problems because of the design of the machine, and the next three chapters will give the necessary information for a machine-language programmer who wants to use the computer in this way. The chapters are, therefore, very specific to the ZX81, and the techniques described in them are not necessarily applicable to other computers.

The chapters are not by any means intended as a general introduction to assembly-language programming with the Z80 chip. A whole book is needed for this, and indeed there are a number of excellent books on the market which teach Z80 mnemonics and the conventional methods of applying them. I will assume, therefore, that the reader is familiar with a standard work on the subject and is looking for the additional information on the design of the ZX81 that he will need if he is to be able to get straight into the task of constructing workable machine-code programs. However, full details will be given of the various subroutines in Chapters 8 and 9, showing how they work, and any special programming techniques that are used in them will be pointed out and explained.

What are the particular problems involved in machine-code programming on the ZX81? Machine-code programs fall into two main groups; those that include their own routines for I/O, that is, for collecting user input and displaying results, and those which are self-contained in the sense that they merely manipulate values which have been passed to them, storing the result in a register or somewhere in memory. The first type of program is machine-specific in the sense that the procedures for putting material onto the screen display, collecting keyboard input, sending data to the cassette port and so on, will always vary from one computer to another because they depend on the architecture of the hardware. Once this architecture has been established, the programmer can always write standard Z80 instructions to fulfil the procedures. One example of this might be the common practice of displaying information on the screen by finding the memory locations of the Display File on the ZX81, and loading byte values into some of these locations so that the equivalent characters will appear on the display. There is nothing wrong with such an approach, but it

repeats work that has already been done by the computer. Since the ZX81 ROM handles all such I/O in the course of implementing BASIC programs, it must contain the necessary routines to do this, and time as well as space can be saved by CALLing the appropriate routine in ROM instead of writing one's own input or output routine.

In practice it turns out that it is sometimes very easy to do this, and sometimes rather tricky. Full details will be given of the method for implementing I/O on the standard 16K ZX81. It might be thought that the second type of machine-code program is easier to handle because the programmer need not concern himself with the architecture of the machine. That is true as far as it goes, but there is not much point in a machine-code program which puts a result in a register or a memory location inaccessible to the user. In practice, such programs will usually be the object of USR calls from BASIC; this means that the result of the calculation, in the BC register pair, can be loaded into a BASIC variable and thus retrieved. The method for using USR needs further explanation as no facility is provided in ZX81 BASIC for passing parameters to the machine-code routine in the first place.

There is another variety of machine-code program which is commonly encountered although it does not display its results directly to the user, and this is the program which operates on in-memory BASIC programs in some way. The most obvious example is the Renumber program, of which there are several on the market, but many other programs of this type still wait to be written. They can be regarded as implementations of statements which are missing from the 8K BASIC of the ZX81 but which would be invaluable to the serious programmer if they were available, and can increase programming efficiency dramatically. Some examples of this kind of program, available for other computers, are:

SQUASH to remove redundant spaces and REM lines from a program

VARLIST which creates a list on the screen of all the variable-names found within the BASIC program

SEARCH finds all occurrences of a given sequence of characters (or tokens) within a BASIC program and lists the line-numbers in which they occur

REPLACE as SQUASH, but replaces all instances of the given sequence with another given sequence

All such machine-code aids require knowledge of the way in which BASIC programs are stored by the ZX81 and the way in which the ROM carries out internal housekeeping, during program execution and editing, by updating various pointers in the operating system extension into low RAM. As these details are unique to the ZX81, programs of this type are in this sense still machine-specific. At any rate, information will be given in the later chapters for any programmer interested in developing utilities of this kind.

There is, however, a limit to the modifications that can be made to the execution of BASIC programs and machine instructions on the ZX81, assuming that the programmer is not planning actually to start cutting tracks on the board. This limit is encountered, for example, if one tries to interrupt the execution of a BASIC program at the end of each line, perhaps in order to provide a TRACE facility, or if one wishes to display individual pixels on the screen. In the first example, the absence of vectors to RAM during BASIC execution will prevent the programmer from interrupting the BASIC program if he is not using an explicit USR call in the program itself. Essentially, this is because the machine was not designed to be expanded beyond 16K, and no "hooks" have been built into the design to enable enhancements to be added to the operating system and BASIC interpreter at a later date. The various ROM routines will vector to addresses which are all located within the ROM itself.

By contrast, the owner of a machine such as the Model 1 TRS-80 or BBC Micro will find an enormous number of these software hooks in the operating system, because these computers are intended to be expanded to include additional hardware and software facilities.

In the second example, it will be discovered that the ZX81 has been hardwired in a non-standard way in order to simplify the board architecture, with the result that some of the Z80 instructions contained in the ROM do unexpected things. The instruction JP (HL) at location 44 hexadecimal actually causes the computer to hang fire for a specified period while a picture is sent to the TV; when the Z80 attempts to execute the instruction, a complicated sequence of events is set in motion on the board, the details of which are not accessible to a programmer who would like to make individual modifications to them.

This point raises another; is the ZX81 really a suitable choice of machine for a machine-code programmer? It is probably not very suitable for someone who wants to gain experience of standard procedures for designing microcomputer systems, as the design is unique to Sinclair Research. It is not very suitable either for someone interested in the very specialised subject of writing ROMable interpreters for machines that have BASIC built-in. Full details of the working of the ZX81 interpreter are still not available, but it is already clear that it is affected at a number of points by the hardware design of the machine and can be fully understood only in this context; it is also clear that it contains a number of errors and examples of awkward coding which one would not necessarily want to imitate. The attitude taken in the present work is that one should retrieve the necessary information on I/O CALLs and so on from the ROM, and then keep well clear of it; the Assembly-language programmer will not normally need to know any more about the ROM than this.

A note on conventions; I will use standard Zilog mnemonics and the conventional way of referring to hex addresses, that is, hexadecimal 44 will be shown as either 44H or 0044H. The Assembler that has been used to generate the listings in this book is

EDTASM+ on a TRS-80. Although one or two Assemblers have appeared for the ZX81, they are still very primitive, and in some cases use non-standard conventions. The reader should be able to "translate" the routines listed here to the format required for his Assembler; EDTASM+ is fairly standard, but the following information should be borne in mind:

The first column of the Assembly listing gives the hex address of the first byte of the instruction.

The second column shows the values, in hex, of the byte or bytes making up the instruction.

The third column is just a line number, used by the Editor that is provided with EDTASM+.

The fourth column may hold an optional label; this may be a jump or CALL vector or just the title of a subroutine, to make it easier to distinguish it from the rest of the coding.

The fifth column holds the instruction itself. Note that this Assembler will represent the instruction SUB A,B as SUB B, not permitting the A register to be quoted in the instruction. The Assembler ZXAS does allow the programmer to enter SUB A,B but will regard it as the instruction SUB A,A and assemble accordingly, giving no warning that this is being done. This can cause many hours of fruitless debugging and has been attributed to a bug in ZXAS, but it is just a non-standard convention compounded by inadequate error-trapping in that Assembler.

The final column holds explanatory comments.

EDTASM+ also permits the use of a number of instructions to the Assembler which are not part of the Z80 instruction set, but are "pseudo-ops". These include:

DEFB This allows the programmer to define the value to be stored at the current memory location (one byte) in the listing. If this pseudo-op is lacking from an Assembler, the one-byte space can be "reserved" during Assembly by using NOP and then the value can be POKEd into the memory location subsequently.

DEFW As DEFB, but defines a two-byte value held as LSB then MSB.

ORG This specifies the first byte of memory that will be occupied by the program on Assembly.

EQU This pseudo-op allows the programmer to equate a numeric value with a label at the beginning of a program, and aids clarity in listings. If, for example, 16388 is equated with the label RAMTOP, instructions such as LD HL, (RAMTOP) can be typed in and will be largely self-documenting.

In the Preface I mentioned that there are (at least) two versions of the ROM on the ZX81, the "old ROM" issued with the original

machine, and the "new ROM" which takes care of bugs in the arithmetic routines and the PAUSE routine. It is not beyond the bounds of possibility that further versions of the ROM will come into circulation in due course - manufacturers commonly make minor alterations to BASIC ROMs without publicising the fact. The following table should, however, allow the reader to establish whether he has one or the other of the known versions:

PRINT PEEK	returns	old ROM	new ROM
3875		253	205
5274		128	42

From the point of view of the Assembly-language programmer, all he has to remember is that one or two of the ROM routines he may want to CALL will be at one address or another, depending on the version of the ROM that he has. We will give listings using the new-ROM addresses, and then show the old-ROM equivalent in brackets.

ROM CALLS

This is a list of the addresses of ROM routines that may usefully be CALLed from one's own machine-language program. They are mostly concerned with I/O; there are other routines in the ROM, concerned with interpretation of BASIC programs, that are usable, but there are few applications that really need them. The information that you will need to be able to use the CALLs includes registers used by the routines, differences between use in FAST or SLOW mode, and so on, and this basic information is also given. Note, however, that it will in practice be necessary to write a routine around the CALL in order to be able to use it safely, in some cases; these encapsulating routines are given in Chapter 8.

The authors of the ZX81 ROM tend to make full use of the registers and in many cases, the phrase "uses all registers" will be encountered in the descriptions, meaning that you cannot safely leave information in a register at the time that you CALL a routine; it will be destroyed on RETurn. In fact there are some routines which will not necessarily use all the registers, depending on the circumstances in which they are CALLed, but it is less complicated in such cases to assume that no registers can safely be used and code accordingly; you can always store information on the stack and then retrieve it later.

Remember, finally, that some of the registers are used by the interrupt routines which implement SLOW mode and that others are supposed to contain fixed values at certain times. All the routines in this book attempt to avoid using these registers at all, but when it proves impossible to avoid using (for example) IX and IY, a special note is made of the fact in the accompanying explanation. At any rate, this peculiarity of the ZX81 (explained on page 167 of

the Sinclair manual) should be taken into account when planning register use.

SLOW

Address: 0F2BH (old ROM: 0F28H)

Uses all registers.

Enters or confirms SLOW mode. You can CALL SLOW at any time.

FAST

Address: 0F23H (old ROM: 0F20H)

Uses no registers (but see Sinclair manual p.167).

Enters or confirms FAST mode. You can CALL FAST at any time.

CHAR

Address: 0010H

Uses alternate register set. It appears that values in the main register set at the time the call is made are unchanged by the call, but this has not been thoroughly checked out.

This of course is one of the Z80's RST addresses, so the idea is to code RST 10H whenever you want to print a single character on the screen, at the current cursor location. The cursor location moves on one space, ready to print the next character.

The Sinclair character code of the character to be displayed can be found in the table in the Sinclair manual, and this value should be loaded into the A register before invoking RST 10H. Registers unaffected by RST 10H include B, H and L. The only control code understood by RST 10H is 76H, which is NEWLINE; if this value is loaded into A, the cursor location will be moved to the beginning of the next line.

Caution: RST 10H will make no investigation of the contents of the memory location within the Display File that it is loading up with the character you have chosen. What this means in practice is that if the current cursor location is one of the 76H's at the end of each line in the Display File, this will be overwritten by your chosen character, which will actually appear on the screen, within the blank margin at the right-hand side of the screen that is normally inaccessible. All may appear to be well at the time, but the Display File now contains one too few 76H's. As soon as the operating system discovers this, as it is bound to do in the course of its periodic housekeeping, you will get a system crash. To use RST 10H, you need to know that the cursor location will be at a permitted position within the Display File; otherwise, you

will need to use the PRNTAT or PRINT$ routines, described below.

CLS

Address: 0A2AH

Uses all registers.

Clears the screen, exactly as the BASIC CLS statement. The screen cursor moves back to the top left position, and locations 16441/2 are updated accordingly; that is to say, 16441 contains 33 decimal, corresponding to the first column, and 16442 contains 24 decimal, corresponding to the first row.

BREAK

Address: 0F46H (old ROM: 0F43H)

Uses all registers.

This will strobe the keyboard to see if BREAK is being pressed. The same thing can be achieved by using a subroutine built around the keyboard STROBE routine (see below), but this is quicker. Note that only one check of the current status of the keyboard is made, lasting only a tiny fraction of a second; you may need to put this call inside a loop. On return from the call, the carry flag is set if the BREAK key was *not* pressed at the time the strobe was carried out; otherwise the carry is reset. This call is principally of value in allowing you to step through short chunks of machine code of your own devising, during debugging, as it allows you (for example) to RETurn to BASIC if things seem to be going wrong.

BYTOUT

Address: 031EH

Uses all registers except HL pair.

This will send one byte to the cassette OUT port, and is used by the BASIC SAVE routine for saving BASIC programs to tape. The byte to be saved is not in a register, but in memory, and HL must point to this memory location at the time that BYTOUT is called.

The computer must be in FAST mode when BYTOUT is called, or the routine will RETurn with no output. BYTOUT incorporates a CALL to BREAK, so there is no need to worry about providing the user with a method of cancelling an instruction to output a large section of memory to tape.

Normally, one would dump a series of consecutive memory addresses to tape, using INC HL (or possibly DEC HL) and decrementing a loop counter. The waveform that is sent to tape and which represents the binary data will be identical to that used for Sinclair BASIC programs. BYTOUT contains a number of timing loops which ensure that the blips on the tape are properly "spaced out", and

there is generally no need for the programmer to worry about the duration of the instructions in his own program, which are executed between each CALL to BYTOUT. It is of course true that there will be a variable delay between each such CALL, depending on the particular instructions that the programmer has put in his program, but the BYTIN routine (see below) will not be thrown out of phase by this. The very slow baud rate (averaging 290 bps) used by BYTOUT is in marked contrast to the speed of execution of Z80 instructions.

There is of course a BYTIN routine in the ROM, but it happens to be unusable by the programmer because it is constructed in such a way that it destroys the return address. The thing to do is to construct a slightly altered version of this routine which RETurns in the usual way, and such a routine will be found in Chapter 8.

PRNTAT

Address: 08F5H

Uses all registers.

This will relocate the screen cursor to any chosen position within the Display File. You have to put a value corresponding to the desired line in the B register, and a value for the column in C. Note that, for this routine, the values are those that you would use for the BASIC PRINT AT statement; that is, the value in B will be 0 if you want the first line and will not exceed 21 (assuming you are keeping to the top 22 lines), and the column value in C will similarly vary from 0 to 31. Load BC with a value that fits these parameters, and CALL PRNTAT.

PRINT$

Address: 0B6BH

Uses all registers.

This will print any sequence of characters that are contained in memory onto the screen. The string starts to appear at the current cursor location; the cursor moves on so that, at the end of the routine, it is at the next unused Display position. You have to put the length of the string in the BC pair, and the memory location of the first character in DE. The values in memory will be converted to characters according to the Sinclair character codes.

It might be expected that this routine would use a straightforward LDIR, but in fact all it does is make repeated use of RST 10H to get the characters onto the screen. The reason for calling this routine instead of using RST 10H within a loop is that it will look out for the end of a Display line; if your string is too long to fit on one line, it just continues on the next, skipping the 76H byte at the end of the line in proper fashion, and avoiding the system crash problem mentioned earlier.

STROBE

Address: 02BBH

Does not use E register; uses HL, BC, A, D.

This routine will make one strobe of the keyboard and return whether a key was being held down at the time or not. It is capable of differentiating between all the possible combinations of multi-key entries that the keyboard allows. What happens is that a value is returned in the HL register pair corresponding to the key depression; however, the value bears no relation to the Sinclair character codes and further processing is necessary to discover what it represents. There are two ways of doing this: one is to have a table showing the correspondence of HL values with characters, and test for each of the values that one is looking out for. Typically, one might ask the user to press either "1" or "2" on the board, and would ignore any other key depression, cycling back to call STROBE again until one of the specified keys is pressed. Such a table is printed on page 162 of Ian Logan's book *Understanding the ZX81 ROM*, and the reader is referred to this. Alternatively, you can use the DECODE routine in ROM (explained below) to convert the HL value for you.

DECODE

Address: 07BDH

Uses all registers.

To use DECODE, transfer the HL value to BC and CALL DECODE. On return, HL points to a memory location which contains a byte corresponding to the code of the character you are after.

In fact the HL value will be an address in the ROM, within the "character table". This is where the ZX81 stores all the characters that it understands and also the expanded BASIC keywords, as well as the one-byte tokens that represent them. This account may seem somewhat confusing, because of the sequence of steps necessary to get hold of the desired character that has been input at the keyboard. However, a routine will be listed in Chapter 8 to explain the whole procedure in stages. There are a couple of pitfalls involved in using these ROM routines, and Chapter 8 will show how to avoid them.

In the meantime, let's just note that there is an alternative way of collecting a keyboard value. Instead of CALLing STROBE, you can look at the contents of locations 16421/2, which will contain the value 65535 if no key is being pressed or some other value if a key depression is currently being detected - just like the value in HL. There is no particular advantage in one technique over the other, and the remainder of this book will use the STROBE CALL.

TAPE FORMAT

A final word on the format used to store ZX81 BASIC programs on tape, which may be of use if you want to operate on ZX81 tapes in interesting ways; you may, for example, want to write a machine-code program to scan a tape looking for program lines in the range 2000-5000 and load them into memory, ignoring the rest of the BASIC program. This is perfectly possible, provided that you are clear about the way in which the program listing, Display File and variables are stored in memory.

When the SAVE statement is encountered, the ZX81 will first of all "output" five seconds of silence. It then outputs the byte values of the name of the tape, which can of course be of variable length, and adds 128 decimal to the final byte (in other words, SETs bit 7). This is followed by a dump of memory locations 16393 to 16508, comprising most of the low RAM operating system extension. Then comes the BASIC program from 16509 onwards (variable length, of course). The 118 decimal byte at the end of the last BASIC program line is followed by another 118 signifying the start of the Display File, and the whole of the File is output. Then, if the location pointed to by locations 16400/01 contains 128 decimal, there are no variables and output stops; otherwise, the variables are output until the routine has reached the memory location pointed to by 16405, the start of "free memory". The BASIC stack, up at the other end of memory, is never saved to tape, hence the warning that one should not SAVE from within a GOSUB call; the return address is stored on the BASIC stack.

The BASIC LOAD routine does not look out for any particular end-of-file byte when reading in a tape; it just notes the value held in 16404/5, which of course was put there near the beginning of the LOAD of the current tape, and ceases looking for data on tape when it has loaded the appropriate number of bytes in.

Chapter 8 Assembly-language subroutines

This chapter provides two kinds of subroutines for a ZX81 Assembly Language programmer. The first kind uses the ROM calls described in the previous Chapter and shows how they can be applied safely. The second type comprises general-purpose subroutines which are likely to be needed by any programmer doing work in Z80 Assembler, and will save him time which might otherwise be spent reinventing the wheel. Some of the routines are quite short and may tempt the BASIC programmer into learning Assembler in order to make his ZX81 a more powerful machine, which in effect is what Assembler has to offer him. However, to repeat the point made earlier, this book does not teach Assembler from scratch although the routines may be found worth studying as examples of applications techniques on the ZX81.

All the routines listed here are assembled with an Origin at location 32000; this is an arbitrary choice and the programmer may choose to locate his versions of them elsewhere in memory. It is, of course, possible to type the routines in by POKEing the decimal equivalents of the machine code to consecutive memory locations, but some sort of Assembler utility will have to be used if serious programming is to be done.

Whatever the location chosen, it will need to be protected by reserving memory. It may be advisable to give a warning at this point: the impression seems to have spread that reserving memory means POKEing values into locations 16388 and 16389. Reserving memory really means relocating the stack pointer, and this can only be done by executing NEW.

When NEW is executed, the ZX81 will investigate the values held at 16388/9 and interpret them as a memory address, then arrange for the stack to build downwards in memory from a location starting just below that memory address. Thus, if you have stored a machine-code routine above this point, it will not be corrupted by values placed on the stack during program execution. Of course, it won't necessarily be corrupted anyway; some routines don't use the stack at all, or it might be that only a few bytes of stack are used which do not overlap your own routine.

A number of programs for the ZX81 have been described as "reserving their own memory". In fact these programs, such as ZXAS, merely load the appropriate values into 16388/9; the stack pointer still points to top of memory until NEW is executed. Another mis-

conception is that memory can be reserved by using the BASIC CLS statement; this results from a misreading of page 177 of the Sinclair manual. CLS will investigate 16388/9, but only to decide what format to use for the Display File; it sets up a full or "minimal" file, as described on page 172 of the manual, depending on the amount of memory it thinks is available.

There may well be a USR call that can be made to reserve memory in the proper sense, but it does not appear to have been found yet. RAND USR 1040 is one of the calls that have been put forward as reserving memory; it doesn't work. Of course, you don't need to find a ROM routine if you just want to move the stack; the Z80 instruction set contains explicit instructions for altering the value in the SP register, and setting up your own temporary stack is a normal programming procedure.

VIDEO DISPLAY

The routine called DISP, shown in Listing 8.1, can be used whenever you want to print a message on the screen. The message needs to be stored in memory as a series of consecutive byte values corresponding to Sinclair character codes. In the Listing, the DEFM facility available under EDTASM+ has been used to make the Assembler generate the appropriate values corresponding to the message "HELLO", and these values can be seen in the Listing occupying locations 7D24H to 7D28H. They are followed by a 76H byte which is used by the routine to signify the end of the message. It had better be explained that DEFM has been used in this Listing merely for explanatory purposes; the byte values shown correspond to standard ASCII whereas Sinclair character codes are non-standard. The assembly-language programmer wanting to include messages in his coding must either look for a ZX81 Assembler which understands Sinclair character codes, or make up his message by looking up the codes in the Sinclair manual letter by letter and POKEing them into memory.

A typical program might contain half a dozen messages, stored one after the other, separated from each other by 76H's. To print a particular message, the programmer must know the location in memory of the first byte of the message, which in the case of Listing 8.1 is 7D24H. He doesn't need to know the length of the message so long as he is sure that it will not take up more than five lines of the screen. He does need to locate the screen cursor to the desired position on the screen, but only if he has special requirements concerning particular rows and columns at which different messages must appear.

The address of the first message byte is then loaded into HL and the routine is CALLed. The routine will make sure that the computer is in SLOW mode and will then ensure that printing a long message will not cause a system crash. It does this by CALLing its own subroutine, SCRCHK. This will investigate location 16442 to find out whether the position of the screen cursor allows for the printing of five lines of message plus a carriage return at the

end of it, while remaining within the top 22 lines of the screen. If not, the screen is cleared so that the new message will appear at the top. When making calculations concerning line positions, one should remember that the twenty-fourth line of the screen corresponds to the value zero in 16442, so one generally wants to avoid printing while 16442 holds a value between zero and two inclusive given that the printing-routine will execute a carriage return at the end of the message.

Listing 8.1

```
7D00            00100           ORG     32000
403A            00110 ROW       EQU     16442
0A2A            00120 CLS       EQU     0A2AH
7D00 E5         00130 DISP      PUSH    HL      ;SAVE MESSAGE START ADDRESS
7D01 CD2B0F     00140           CALL    0F2BH   ;SLOW
7D04 CD1D7D     00150           CALL    SCRCHK
7D07 E1         00160           POP     HL      ;MESSAGE ADDRESS TO HL
7D08 E5         00170           PUSH    HL      ;AND COPY...
7D09 D1         00180           POP     DE      ;...TO DE
7D0A 010000     00190           LD      BC,0    ;INITIALISE MESSAGE LENGTH COUNTER
7D0D 7E         00200 LOOP      LD      A,(HL)  ;FIRST CHARACTER TO A
7D0E 23         00210           INC     HL
7D0F FE76       00220           CP      76H     ;END OF MESSAGE?
7D11 2803       00230           JR      Z,FINAL ;JUMP IF SO
7D13 03         00240           INC     BC      ;ELSE ADD 1 TO MESSAGE LENGTH
7D14 18F7       00250           JR      LOOP
7D16 CD6B0B     00260 FINAL     CALL    0B6BH   ;PRINT$
7D19 3E76       00270           LD      A,76H   ;EXECUTE...
7D1B D7         00280           RST     10H     ;...NEWLINE
7D1C C9         00290           RET
7D1D 3A3A40     00300 SCRCHK    LD      A,(ROW) ;CHECK SCREEN STATUS
7D20 FE07       00310           CP      7       ;ROOM FOR 5 LINES?
7D22 DC2A0A     00320           CALL    C,CLS   ;IF NOT, MAKE ROOM
7D25 C9         00330           RET
7D26 48         00340           DEFM    'HELLO'
7D27 45
7D28 4C
7D29 4C
7D2A 4F
7D2B 76         00350           DEFB    76H
0000            00360           END
00000 TOTAL ERRORS
CLS     0A2A
DISP    7D00
FINAL   7D16
LOOP    7D0D
ROW     403A
SCRCHK  7D1D
```

DISP will use the PRINT$ routine in ROM, described in the previous chapter. In order to do this it needs to know the message length, and lines 200-250 of the Listing use a loop to build up the length of the message in BC, by checking consecutive memory locations to see if the end-of-message byte has been reached yet.

The start address of the message has already been saved in DE, and so the PRINT$ routine can be CALLed in line 260. After this, a carriage return is implemented and execution RETurns from DISP. Many programs will be of the type which print a series of messages during execution, each one beginning at the start of a new line, and if the programmer simply wants to do this, DISP will take care of all the details without requiring him to worry about the current screen cursor location before each CALL to DISP. If, on the other hand, there are short messages which must appear near the bottom of the screen, the value used in line 310 can be reduced to permit this.

LINE INPUT

If the user of a program is required to type in more than one character at a time, the program must be able to store such a string of input somewhere so that it can operate on it. Of course, a machine-code program may run so fast that it can perform any necessary operations on each key depression before the user has time to hit the next key, and programming of this kind will be found in most machine-code arcade games. There are, however, many occasions when it is inconvenient or impossible to structure code in this way, and the normal approach is to use the input buffer to store the series of characters. This is an area of RAM set aside for such data, and the bad news in the case of the ZX81 is that it doesn't exist, nor is there a ROM routine which would be able to use it.

Multi-character or line input from the user is handled by the ROM in a variety of ingenious ways, none of them of much use to the assembly-language programmer. The solution is to build up a routine which can handle keyboard input of this kind and to include in it some otherwise unused memory locations where the input can be stored, to replace the missing input buffer.

There is a little more to it than that. The routine needs to be able to display the input on the screen character by character; it follows that there must be some control over the number of characters the user is allowed to input, otherwise he might provoke a system crash just by typing past the end of a line on the screen. The ZX81 keyboard allows the user to type in multi-character BASIC keywords with single strokes; most programs that the assembly-language programmer will want to write must positively prevent this happening, as the only input needed will be from the alpha-numeric character set, and the appearance of expanded BASIC tokens on the screen as multi-character keywords will mess up the display. Although the routine will allow input characters only up to a certain maximum, it should nevertheless allow the user to terminate input before this maximum of keystrokes is reached, or it will be far too inflexible for general use. Finally, something has to be done about the latent "keybounce" problem in the ZX81 keyboard. An assembly-language routine which continually strobes the keyboard looking for input will detect a key depression easily enough, but unless some extra routine is added, it will continue to detect the

same key depression over and over again until the user lifts his finger from the key. There are several ways of coping with this, the one used in Listing 8.2 being a simple delay loop; some such routine is necessary to avoid an unwanted auto-repeat or "bounce" on keyboard input.

Listing 8.2

```
7D00          00100          ORG     32000
07BD          00110 DECODE   EQU     07BDH
7D00 E5       00120 GETSTG   PUSH    HL      ;SAVE START ADDRESS OF INPUT BUFFER
7D01 54       00130          LD      D,H     ;CLEAR...
7D02 5D       00140          LD      E,L     ;...BUFFER...
7D03 13       00150          INC     DE      ;...OF...
7D04 3600     00160          LD      (HL),0  ;...PREVIOUS...
7D06 EDB0     00170          LDIR            ;...INPUT
7D08 E1       00180          POP     HL
7D09 060F     00190          LD      B,15    ;MAXIMUM INPUT LENGTH
7D0B E5       00200 LOOPR    PUSH    HL
7D0C C5       00210          PUSH    BC
7D0D CD1B7D   00220          CALL    INPUT   ;COLLECT NEXT VALID CHARACTER
7D10 C1       00230          POP     BC
7D11 E1       00240          POP     HL
7D12 FE76     00250          CP      76H     ;END OF INPUT?
7D14 C8       00260          RET     Z       ;RETURN IF SO
7D15 D7       00270          RST     10H     ;ELSE ECHO TO SCREEN
7D16 77       00280          LD      (HL),A  ;AND STORE IN BUFFER
7D17 23       00290          INC     HL      ;HL POINTER TO NEXT FREE BUFFER LOCATION
7D18 10F1     00300          DJNZ    LOOPR
7D1A C9       00310          RET             ;AUTO-RETURN IF USER INPUTS MAXIMUM-LENGTH STRING
              00320 ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
7D1B CD4F7D   00330 INPUT    CALL    STROBE  ;GET KEY DEPRESSION
7D1E 2B       00340          DEC     HL      ;BACK TO KEYHIT VALUE
7D1F 7D       00350          LD      A,L
7D20 FEFF     00360          CP      255     ;SHIFT KEY ONLY?
7D22 28F7     00370          JR      Z,INPUT ;IGNORE IF SO
7D24 CD307D   00380          CALL    ALPNUM  ;ELSE ANALYSE FURTHER
7D27 30F2     00390          JR      NC,INPUT;NOT IN PERMITTED CHARACTER SET, IGNORE AND TRY AGAIN
7D29 44       00400          LD      B,H
7D2A 4D       00410          LD      C,L
7D2B CDBD07   00420          CALL    DECODE  ;WITH KEYHIT VALUE IN BC
7D2E 7E       00430          LD      A,(HL)  ;CHARACTER TO A FROM ROM TABLE
7D2F C9       00440          RET
              00450 ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
              00460 ;PERMITTED CHARACTERS HAVE 0FEH OR 07FH IN L OR HAVE ONE ZERO BIT IN H
7D30 7D       00470 ALPNUM   LD      A,L
7D31 FEFE     00480          CP      0FEH
7D33 2818     00490          JR      Z,LAST  ;OK
7D35 FE7F     00500          CP      07FH
7D37 2814     00510          JR      Z,LAST  ;OK
              00520 ;NOW COUNT UP NUMBER OF 1-BITS IN INPUT BYTE
7D39 7C       00530          LD      A,H
7D3A 1600     00540          LD      D,0
7D3C 0608     00550          LD      B,8
7D3E 17       00560 LOOPQ    RLA
```

```
7D3F DC4B7D   00570          CALL    C,INC
7D42 10FA     00580          DJNZ    LOOPQ
7D44 7A       00590          LD      A,D
7D45 FE07     00600          CP      7
7D47 2804     00610          JR      Z,LAST  ;OK IF EXACTLY SEVEN 1-BITS
              00620 ;AT THIS POINT BYTE HAS FAILED ALL TESTS
7D49 B7       00630          OR      A       ;CLEAR CARRY = REJECT CHARACTER
7D4A C9       00640          RET
7D4B 14       00650 INC      INC     D       ;ADD ONE TO COUNT OF 1-BITS
7D4C C9       00660          RET
7D4D 37       00670 LAST     SCF             ;SET CARRY = PERMITTED CHARACTER
7D4E C9       00680          RET             ;WITH C SET/RESET SHOWING WHETHER PERMITTED CHARACTER
              00690 ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
7D4F CDBB02   00700 STROBE   CALL    02BBH   ;LOOK AT KEYBOARD
7D52 23       00710          INC     HL      ;ADD ONE TO KEYHIT VALUE
7D53 7C       00720          LD      A,H
7D54 B5       00730          OR      L
7D55 28F8     00740          JR      Z,STROBE;ZERO MEANS NO KEY BEING DEPRESSED, TRY AGAIN
7D57 0E2D     00750          LD      C,45    ;DELAY = 1/2 SEC.
7D59 06FF     00760 OUTER    LD      B,255
7D5B 10FE     00770 DELAY    DJNZ    DELAY
7D5D 0D       00780          DEC     C
7D5E C8       00790          RET     Z       ;WITH KEYHIT VALUE + 1 IN HL
7D5F 18F8     00800          JR      OUTER
0000          00810          END
00000 TOTAL ERRORS
ALPNUM  7D30
DECODE  07BD
DELAY   7D5B
GETSTG  7D00
INC     7D4B
INPUT   7D1B
LAST    7D4D
LOOPQ   7D3E
LOOPR   7D0B
OUTER   7D59
STROBE  7D4F
```

The line input routine called GETSTG is shown in Listing 8.2 and consists of a number of subroutines performing different tasks, to overcome the difficulties described above. Probably the best thing to do is to start in the middle of the complete routine and show how the various subroutines in it function; in any case many of them can be used independently, in your own programs. Then we can see how the routine as a whole is constructed.

The subroutine STROBE at lines 700-800 will scan the keyboard using the ROM routine at 02BBH, described in Chapter 7. If no key is being depressed at the time, the value 65535 is returned in the HL register pair. To detect this, it's convenient to increment HL by one and test for zero; the lines from 720 to 740 show the conventional way of testing a register pair for zero on the Z80. The subroutine keeps cycling around until a key is hit, then waits for half a second before returning with whatever value is in HL. The nested loop at 750-800 is, again, a conventional way of implement-

ing fairly short delays of this kind, and prevents the same depression being read several times before the user has time to remove his finger from the keypad; half a second may seem excessive, but the value in line 750 can always be altered to give a different period.

Now that the key depression has been translated into a value in HL, the DECODE routine in ROM will translate it again into a character code. However, it would be disastrous to CALL DECODE with this value, without further processing. This is because the value in the L register may be 255, and, if so, the ROM routine will go into an infinite internal loop; it does not expect to be called with that particular value in L. There is also the question of undesirable input outside a permitted range; the GETSTG routine will allow input of the letters A-Z, the numbers 0-9, and the punctuation symbols on the bottom row of the keypad, accessible through the Shift key. It will not accept other key combinations, and the subroutine which tests key depressions for admissibility is called ALPNUM.

ALPNUM is located at lines 460-680. It turns out that three tests have to be carried out on the HL value to establish whether it is in the permitted range; line 460 gives further details. A jump to LAST is made if the L register contains either of the two values which show that the value is permissible. If not, the H register has to be tested to see whether it holds exactly one zero bit, and this is done in lines 530-610. The method used is to transfer H to A and perform eight left-rotates, the Carry being set if the bit that falls out of the byte is a one. The little subroutine INC will tot up the number of times this happens and send control to LAST if the answer is 7. LAST will set the Carry flag to show that the value is OK, otherwise the Carry is reset.

Let us now look at the INPUT subroutine, lines 330-440. It begins with a CALL to STROBE, and then decrements the HL value; the reader will instantly recall that we incremented HL in STROBE, so we need to adjust it back again. We now have a special test to see if the Shift key (and only the Shift key) is the one that is being depressed; if so we reject it and try again. Otherwise we CALL ALPNUM, and on return (line 390) test the Carry to see if the character is in the permitted range. If it is, the HL value has to be transferred to the BC register pair, and we can then safely CALL DECODE.

On return from DECODE, there will be a value in HL pointing to the address of the character in the ROM character table. Line 430 therefore transfers the correct Sinclair character code to the A register. At long last we are in a position to write a routine to collect key depressions in A and store them in our own input buffer. First, though, we should remember to delete any previous input from the buffer that may be left over from some other use of it - as a matter of fact it may be used to store all kinds of temporary values generated by other parts of our programs. It may not be necessary to carry out this deletion, depending on the way in which your program examines user input in the buffer, but at

any rate the GETSTG Listing commences with a procedure at lines 130-170 to do this; it writes zeros to memory locations commencing with the location pointed to by HL, and continuing until BC has been decremented to zero by the LDIR instruction. It should be noted, therefore, that the address of the start of your buffer should be in HL at the time you CALL GETSTG, and (if you are intending to use GETSTG without amendment) you will need to store the length of the buffer in BC.

Once the buffer is clear, GETSTG still needs to know the start address (in HL) and the maximum permitted length of input, which must be stored in B. The process of collecting input is then carried out in lines 200-300, by repeated CALLs to INPUT. Each time we return from INPUT, a valid character code is in A. NEWLINE is one of the characters that the routine permits, so this is how we can detect premature termination of input by the user, and line 260 causes a RETurn from the GETSTG routine if NEWLINE is hit. Otherwise the character is printed on the screen; RST 10H is used, so the screen cursor must be at a pre-calculated position where it is safe to print the characters. The character is stored in the buffer and another CALL to INPUT is made, but if enough characters have been collected to fill the buffer, the routine automatically RETurns without any further action by the user.

You may want to abstract some of the subroutines in GETSTG for your own programs, or use it with little alteration. At any rate it provides a ZX81 assembly-language programmer with a fairly powerful tool, as there is a limit to the range of programs that can be written without some kind of line input routine, and it would be too restricting to implement user input in BASIC with frequent jumps to machine code and back again.

READING TAPES

This is an explanation of the subroutine BYTIN, shown in Listing 8.3, and intended to be used to complement BYTOUT, the ROM routine which sends a byte to the cassette out port. As a matter of fact there is not a lot of explanation in the Listing; a full account of the way BYTIN works would also require a detailed description of the waveform format created on tape by BYTOUT. Basically, lines 110-220 cause the computer to look to the in port for any kind of signal; in practice, "signals" of all kinds are constantly being received, mostly transients of very short duration which are not part of a tape recording. These lines contain a rudimentary filter to cut out some of this noise, although it is still "echoed" to the screen as the familiar diagonal black bands. As soon as something comes in which looks more like real data, it is passed to GETBIT at 230-370 to analyse whether it is a one or a zero; the ZX81 tape format is rather unusual in that all data consists of bunches of sine waves of different duration. Absence of signal does not signify a zero bit, as on some other systems, but merely the absence of data.

Listing 8.3

```
7D00            00100           ORG     32000
7D00 0E01       00110 BYTIN     LD      C,1
7D02 0600       00120 LOOPA     LD      B,0
7D04 3E7F       00130 LOOPB     LD      A,7F
7D06 DBFE       00140           IN      A,(0FEH);STROBE IN PORT
7D08 D3FF       00150           OUT     (0FFH),A;ECHO TO SCREEN
7D0A 1F         00160           RRA
7D0B D2A203     00170           JP      NC,03A2H;BREAK
7D0E 17         00180           RLA
7D0F 17         00190           RLA
7D10 3805       00200           JR      C,GETBIT
7D12 10F0       00210           DJNZ    LOOPB
7D14 C3007D     00220           JP      BYTIN
7D17 1E94       00230 GETBIT    LD      E,094H
7D19 061A       00240 LOOPC     LD      B,01AH
7D1B 1D         00250 LOOPD     DEC     E
7D1C DBFE       00260           IN      A,(0FEH)
7D1E 17         00270           RLA
7D1F CB7B       00280           BIT     7,E
7D21 7B         00290           LD      A,E
7D22 38F5       00300           JR      C,LOOPC
7D24 10F5       00310           DJNZ    LOOPD
7D26 2004       00320           JR      NZ,SKIP
7D28 FE56       00330           CP      056H
7D2A 30D6       00340           JR      NC,LOOPA
7D2C 3F         00350 SKIP      CCF
7D2D CB11       00360           RL      C
7D2F 30D1       00370           JR      NC,LOOPA
7D31 C9         00380           RET             ;FROM BYTIN WITH BYTE IN C
0000            00390           END
00000 TOTAL ERRORS
BYTIN   7D00
GETBIT  7D17
LOOPA   7D02
LOOPB   7D04
LOOPC   7D19
LOOPD   7D1B
SKIP    7D2C
```

GETBIT assesses the incoming signal by creating a value in the E register which corresponds to signal duration. There are two ranges of values which may be in E and which correspond to the "one" and "zero" signals; in this case the bit is loaded into the C register. If the E value is outside both these ranges, the signal is ignored and GETBIT starts looking for another one. When eight bits have been loaded into C, the routine RETurns and the programmer can extract the byte from C without difficulty.

The routine is actually quite precise about the data that it will accept as valid and rejects poor-quality input, hence the reputation of the ZX81 as a "difficult loader". Unfortunately, if the quality is too bad, the machine just gives up and returns the

user to the inverse-K mode without giving any diagnostic information. At any rate the programmer doesn't need to worry about any of this; he can just CALL BYTIN whenever he wants to read a byte from tape. BYTIN will loop round until the data starts coming in, then read a byte and RETurn. It is the programmer's responsibility to keep up with the baud rate in the sense that there must not be too much coding between each CALL to BYTIN, or it's possible that the routine will miss the beginning of the next byte on tape. In practice, as mentioned in the previous chapter, the baud rate is so low that there is time for scores of instructions to be executed between each byte.

BYTIN is coded to match BYTOUT exactly and will read any tape that can be read by the BASIC LOAD routine. There is, of course, nothing to stop you creating your own tape signal format, in which case you can opt for a much higher baud rate, but then you will have to write your own signal-output routine to replace BYTOUT. It seems preferable to stick to the Sinclair format as this means that you can use BYTIN to read BASIC tapes; you have to be clear as to how many bytes of data are going to be on the tape and how to discover this while reading it in, and some information about this is given in Chapter 9.

As a final point, you should be in FAST mode when you CALL BYTIN, and you can safely use the HL register pair, which will be preserved during the CALL. In practice it is almost certain to be used to hold a pointer to a memory address and to be INCremented by one between each CALL.

NUMBER CONVERSION: from binary

When a user inputs characters we can store them and later analyse them comparatively easily, but when he is going to be typing in numeric values we are going to have to do some conversion work as well. We may be able to persuade him to use hex rather than decimal, but either way we will have to convert the value to binary so that it can be transferred to a register or register pair. Similarly, if we want to display numeric values on the screen, we will always be starting with a value stored in binary form and will have to convert it to something more comprehensible before printing it out. Number conversion routines are needed by all programmers on all computers (even an arcade game will display your current score at the top of the screen), and the routines provided in Listings 8.4 and 8.5 are intended to be useful to a ZX81 programmer who doesn't want to work them out for himself.

Generally, decimal is much more awkward to deal with than hex. In Listing 8.4, the routine BINDEC in lines 110-300 uses a table of decimal values (lines 450-490) to make things easier. On entry to the routine, the number that we want to convert to decimal representation is stored in the HL pair as an unsigned binary number. Suppose that this value is 32767. The routine will collect the first value (10000) from DECTBL and subtract it from the original value until the number goes negative, incrementing A each

Listing 8.4

```
7D00            00100           ORG     32000
7D00 11337D     00110 BINDEC    LD      DE,DECTBL
7D03 1A         00120 BD1       LD      A,(DE)  ;FIRST...
7D04 4F         00130           LD      C,A     ;...DECIMAL...
7D05 13         00140           INC     DE      ;...NUMBER...
7D06 1A         00150           LD      A,(DE)  ;...TO...
7D07 47         00160           LD      B,A     ;...BC
7D08 AF         00170           XOR     A
7D09 B7         00180           OR      A
7D0A ED42       00190 BD2       SBC     HL,BC   ;START SUBTRACTING DECIMAL NUMBER FROM BINARY NUMBER
7D0C 3803       00200           JR      C,BD3   ;HAS BECOME A MINUS VALUE
7D0E 3C         00210           INC     A
7D0F 18F9       00220           JR      BD2
7D11 09         00230 BD3       ADD     HL,BC   ;RESTORE TO POSITIVE VALUE
7D12 C69C       00240           ADD     A,156   ;CONVERT TO INVERSE 0-9
7D14 D7         00250           RST     10H
7D15 79         00260           LD      A,C     ;TEST FOR END OF DECTBL
7D16 FE01       00270           CP      1
7D18 C8         00280           RET     Z       ;RETURN IF SO
7D19 13         00290           INC     DE
7D1A 18E7       00300           JR      BD1     ;ELSE KEEP GOING
                00310 ;**********************************************
7D1C 23         00320 BINHEX    INC     HL      ;POINT TO MSB
7D1D CD287D     00330           CALL    GENIUZ
7D20 2B         00340           DEC     HL      ;POINT TO LSB
7D21 CD287D     00350           CALL    GENIUZ
7D24 3EAD       00360           LD      A,173   ;SINCLAIR INVERSE-H
7D26 D7         00370           RST     10H
7D27 C9         00380           RET
7D28 CD2B7D     00390 GENIUZ    CALL    GENIUS
7D2B ED6F       00400 GENIUS    RLD             ;4 BITS TO BITS 3-0 OF A
7D2D E60F       00410           AND     0FH     ;ZAP BITS 7-4 OF A
7D2F C69C       00420           ADD     A,156   ;CONVERT TO INVERSE 0-F
7D31 D7         00430           RST     10H
7D32 C9         00440           RET
7D33 1027       00450 DECTBL    DEFW    10000
7D35 E803       00460           DEFW    1000
7D37 6400       00470           DEFW    100
7D39 0A00       00480           DEFW    10
7D3B 0100       00490           DEFW    1
0000            00500           END
00000 TOTAL ERRORS
BD1     7D03
BD2     7D0A
BD3     7D11
BINDEC  7D00
BINHEX  7D1C
DECTBL  7D33
GENIUS  7D2B
GENIUZ  7D28
```

time until (not including) the time this happens. A therefore contains the value 3, which is the first number we want to display on the screen; the routine jumps to BD3 (line 230), adjusts the HL value back to 2767, adjusts A so that its value now corresponds to the Sinclair character code for "3", and prints it on the screen. RST 10H is used to do this, as in the binary-to-hex conversion shown in the second half of the listing, so the usual precautions must be taken to ensure that the character appears in the correct location on the screen. Note also that the routine prints leading zeros; a decimal value of 456, for example, will appear as "00456".

The routine then cycles back to BD1 in line 120 to deal with the remaining four decimal characters. When the last of them is printed, the C half of the BC register pair will contain a one, as this was the last decimal value to be loaded from DECTBL, so a RETurn is then made from the routine. The routine will handle any value from zero to 65535 and, like most routines which use table look-up techniques, is very fast, although longer than absolutely necessary; an entirely different algorithm could be used to compress it into about half its length, but there is no other advantage in doing this.

The binary-to-hex routine in lines 320-440 of Listing 8.4 is more straightforward, as hex notation is a little closer to the number format that the computer understands. This routine uses a rather elegant technique involving the RLD instruction. This is normally reserved for BCD arithmetic, which tends to be underutilised on the Z80. The best way to regard Z80 instructions, however, is in terms of what they actually do rather than the purpose that their designers may have had in mind, and RLD is a useful way of moving groups of four bits around between registers.

To use the routine, a value must be placed in HL which will be a pointer to a memory location containing the binary value that we want to convert. Again, the routine will deal with unsigned integers between zero and 65535. It expects the value to be stored in two bytes in the format most common on the Z80, that is, LSB followed by MSB. We will want to print this on the screen as four characters representing a hex value, and the first two characters will correspond to the contents of the memory location holding the MSB; the routine therefore begins by incrementing the HL pointer so that it points to that address.

The actual work is done by the subroutine GENIUS; conceited labels of this kind are the privilege of the assembly-language programmer who has done a neat bit of coding. GENIUS must be executed exactly twice before a RETurn is made to BINHEX, and a useful way of doing this is to use the call-and-fall-through technique shown in lines 390 and 400. The RLD instruction will collect the four high-order bits of the MSB in memory and transfer them to the low-order bits of A; at the same time it moves the four low-order bits of the MSB to the high-order bit positions of the same memory location, so that another RLD will be able to collect them next time.

The four bits that have been moved to A represent the first hex character to be printed on the screen, so all we have to do is mask out the high order bits of A in line 410 and then convert the remaining value to correspond with the Sinclair character set. Even this is easy because the authors of the ROM have thoughtfully arranged that the letters A-F will follow 0-9 in their character set; we can alter our number to fit into this range by adding 156 decimal to it. This will in fact print the character in inverse video, but a different constant can be chosen for black-on-white representation.

After two executions of GENIUS, both nibbles of the MSB have been dumped to the screen; line 340 will decrement HL to point to the LSB, and the same procedure is repeated. Finally, line 360 arranges for the letter "H" to appear immediately after the hex number on the screen.

NUMBER CONVERSION: to binary

Listing 8.5 shows routines for converting from decimal, and then from hex, to binary, within the same number ranges as were used for Listing 8.4. DECBIN is fairly intricate and the reader

Listing 8.5

```
7D00          00100         ORG     32000
7D00 CD230F   00110 DECBIN   CALL    0F23H   ;FAST
7D03 D5       00120         PUSH    DE      ;ADDRESS OF FIRST DECIMAL DIGIT
7D04 FDE1     00130         POP     IY      ;TO IY
7D06 DDE5     00140         PUSH    IX      ;SAVE IX VALUE
7D08 210000   00150         LD      HL,0
7D0B DD21607D 00160         LD      IX,DECTBL
7D0F 0E00     00170         LD      C,0
7D11 FD4600   00180 LOOP3   LD      B,(IY)  ;DECIMAL DIGIT TO B
7D14 04       00190         INC     B       ;ONE MORE THAN ACTUAL VALUE
7D15 FD23     00200         INC     IY      ;POINT TO NEXT DIGIT
7D17 DD7E00   00210         LD      A,(IX+0)
7D1A FE01     00220         CP      1       ;LAST DECTBL NUMBER?
7D1C 2002     00230         JR      NZ,NOT  ;NO, SKIP MARKER
7D1E 0E01     00240         LD      C,1     ;END-OF-ROUTINE MARKER
7D20 DD5E00   00250 NOT     LD      E,(IX+0)
7D23 DD5601   00260         LD      D,(IX+1)
7D26 DD23     00270         INC     IX
7D28 DD23     00280         INC     IX
7D2A 19       00290 DEC2    ADD     HL,DE   ;ADD DECIMAL NUMBER FROM DECTBL TO CURRENT TOTAL
7D2B 10FD     00300         DJNZ    DEC2
7D2D AF       00310         XOR     A
7D2E ED52     00320         SBC     HL,DE   ;ADJUST FOR EXTRA ADD IN LINE 190
7D30 79       00330         LD      A,C
7D31 FE01     00340         CP      1       ;END-OF-ROUTINE MARKER?
7D33 2802     00350         JR      Z,LASTD
7D35 18DA     00360         JR      LOOP3
7D37 DDE1     00370 LASTD   POP     IX      ;END OF ROUTINE, RESTORE IX
7D39 FD210040 00380         LD      IY,4000H;RESTORE IY VALUE
```

```
7D3D E5       00390         PUSH    HL
7D3E CD2B0F   00400         CALL    0F2BH   ;SLOW
7D41 E1       00410         POP     HL      ;HL HOLDS BINARY EQUIVALENT
7D42 C9       00420         RET
              00430 ;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
7D43 D5       00440 HEXBIN  PUSH    DE      ;ADDRESS OF FIRST HEX DIGIT...
7D44 E1       00450         POP     HL      ;...TO HL
7D45 7E       00460         LD      A,(HL)
7D46 CB27     00470         SLA     A       ;MOVE...
7D48 CB27     00480         SLA     A       ;...TO BITS...
7D4A CB27     00490         SLA     A       ;...7-4...
7D4C CB27     00500         SLA     A       ;...OF A
7D4E 23       00510         INC     HL
7D4F 86       00520         ADD     A,(HL)  ;NEXT DIGIT TO BITS 3-0
7D50 57       00530         LD      D,A     ;SAVE IN D
7D51 23       00540         INC     HL      ;POINT TO THIRD DIGIT
7D52 7E       00550         LD      A,(HL)
7D53 CB27     00560         SLA     A       ;REPEAT PROCEDURE
7D55 CB27     00570         SLA     A
7D57 CB27     00580         SLA     A
7D59 CB27     00590         SLA     A
7D5B 23       00600         INC     HL
7D5C 86       00610         ADD     A,(HL)
7D5D 5F       00620         LD      E,A     ;TRANSFER TO E
7D5E EB       00630         EX      DE,HL   ;BINARY EQUIVALENT IN HL ON RETURN
7D5F C9       00640         RET
7D60 1027     00650 DECTBL  DEFW    10000
7D62 E803     00660         DEFW    1000
7D64 6400     00670         DEFW    100
7D66 0A00     00680         DEFW    10
7D68 0100     00690         DEFW    1
0000          00700         END
00000 TOTAL ERRORS
DEC2    7D2A
DECBIN  7D00
DECTBL  7D60
HEXBIN  7D43
LASTD   7D37
LOOP3   7D11
NOT     7D20
```

may prefer just to use it rather than spend a lot of time studying it. It has to use IX and IY, and so a number of precautions are taken in the first few lines; the routine executes in FAST mode as SLOW mode requires these registers for the display routines, the IX value is saved on the stack and, at the end of the routine, the value 4000H is put back in IY. On entry to the routine, DE contains the address of the first of the decimal digits to be converted. These values must be "prepared" correctly: leading zeros must be added to the decimal number so that it always occupies five consecutive bytes in memory. In practice, DE will normally point to the start of an input buffer where user input has been stored. These memory locations must contain the actual values of the decimal numbers rather than their representation as Sinclair character codes; for example, if the number is 32767, the address pointed to by DE must contain the value 3, and this value is

transferred to B. DECTBL is again used, and the technique is similar to BINDEC: the value 10000 is repeatedly added to HL to build up the total. B is decremented each time to ensure that the correct number of loops is performed. The routine then obtains the next character from the input buffer and the next entry in DECTBL, and the process is repeated. At the end of the routine, the cumulative total is in HL.

HEXBIN in lines 440-640 uses variations on the techniques employed in the other routines. Again, the four hex digits are at a memory address pointed to by DE, in four consecutive bytes, and have been prepared so that each byte contains a value from zero to 15. The address is copied to HL and the first byte value loaded to A. This value will be the More Signicant nibble (MSN) of the MSB, and it is destined to be saved in the MSN of the D register. Four left shifts are performed to get the value into the MSN of A, and then the second of the four hex digits can be collected from the next location in the input buffer and added into A by line 520, occupying the low-order bit positions. The contents of A are then copied straight into D. Lines 540-620 repeat the procedure with the third and fourth digits, putting them in E. Line 630 merely swaps the completed binary value into HL so that it will be a little more convenient to access on RETurn from the routine.

The reader may readily see that the HEXBIN routine is nearly twice as long as it needs to be; two sections of coding are duplicated and could be replaced by CALLs to a common subroutine. Anyone who would like to develop his assembler skills is invited to rewrite it in a more compact form. The attitude often taken in these matters, however, is that when a machine-code routine actually works, it's already quite good enough, and no further time need be spent on it.

The routines in this chapter give the programmer the beginnings of a library of subroutines which will enable him to tackle a range of projects. They are not very exciting in themselves, but in the final chapter we will consider some more interesting applications programs which can achieve valuable results, and show what considerations need to be taken into account when planning them.

Chapter 9 Assembly-language applications programs

This chapter will illustrate some applications of the routines discussed in Chapter 8 and point out some interesting assembly-language applications for the ZX81. In general, many programs which ought to be available for this computer have not yet appeared on the market, and this includes programming utilities which can speed up development work on BASIC programs. Utilities which have been published include renumbering facilities, block line deletion and the equivalent of the VERIFY command on the Spectrum, so it would be a good idea to look for something that hasn't been done yet.

HYBRID PROGRAMMING

This term refers to the practice of interleaving BASIC coding with calls to machine-code routines *via* the USR statement. It is possible in theory to replace any given BASIC statement with the equivalent in machine-code, but it's considerably more difficult with some statements than others. In practice there are two compelling reasons to write programs of this kind: to speed up BASIC program execution, or to do something that can't easily be done in BASIC at all.

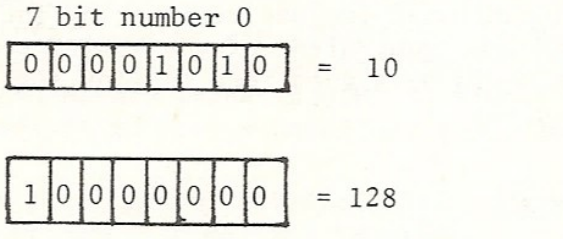
One of the tasks that is difficult to perform in ZX81 BASIC is the manipulation of individual bits in a byte. The need for this is most likely to arise if the programmer has a lot of data to store and wants to compact it into the minimum of memory. Suppose that we wanted to store 5000 values, each in the range zero to 13. ZX81 BASIC only permits single-precision numeric variables, each occupying five bytes, and a relatively enormous amount of memory would be needed to store the data in a numeric array. It would be much "cheaper" to POKE the values into individual bytes, but this would still take twice as much memory as necessary; a nibble (four bits) is capable of storing any value between zero and 15, so it should be possible to store two values in each byte and hold all the data in 2500 bytes.

In order to do this, we need a way of constructing a value between zero and 255 which represents two of the items of data, and then a way of retrieving the individual items. Constructing the byte value is, in this case, easy; multiply the first item by 16 and add the second. It would however be more complicated, in BASIC, to build up a byte value representing three items, if, for

example, the first one occupied bits 7-6, the second one was held in bits 5-3, and the third in bits 2-0. We need a way of modifying individual bits while leaving the others unchanged. To extract these data items, we need to be able to mask out specified bits so that the ones containing the item we are after can be copied out to a variable value.

The logical operators AND and OR in the Z80 instruction set perform exactly this function; AND will mask out unwanted bits while OR will write values to specified bit locations in a byte. The illustration of the OR instruction in Figure 9.1 shows how this works; it should be borne in mind that to use this technique, the bits to be modified should all be zero to start with, as OR is not capable of forcing a zero into a bit location holding a one.

Figure 9.1



10 OR 128 writes a 1 to
bit 7 of the byte, giving
a result of 138

AND and OR are available as statements in ZX81 BASIC, but have entirely different definitions. They can operate only on complete five-byte values rather than the individual bits of a byte, and are useless for the particular purpose we have in mind. By contrast, an assembly-language routine to execute bitwise logical operations is very straightforward to write, and is a suitable application for hybrid programming techniques. We will write a program to implement logical-OR in BASIC with a USR call to such a routine; this will also illustrate one of the problems of USR on the ZX81, that it does not allow any easy way of passing arguments from BASIC to the routine.

The BASIC program, as well as the machine-code routine, will have to be tailor-made to our particular requirements. In this case we'll assume that we want to take two four-bit values and store them in one byte; the machine-code routine will not only OR the two values together but will perform the bit-shifting necessary to ensure that each one occupies the required four-bit field in the byte. As mentioned earlier, this could be done by multiplication in BASIC, but it would slow down the program considerably; the BASIC

ROM multiplication routine is a general-purpose routine for dealing with all possible cases of multiplication that may arise in a BASIC program during execution, and is very slow. All we want to do is shift bits, and the Z80 instructions allow us to do this directly.

In the BASIC program shown below, the two values are shown as ARG1 and ARG2, where ARG2 is to end up in bits 4-7 of a memory location and ARG1 should occupy bits 0-3. The location used in the BASIC program is identified as 28000, but normally you would be writing to a series of such storage locations. It may be questioned why the actual storage of the OR'd value is done by a BASIC POKE rather than by the machine-code routine. It would be possible to pass the memory address to the routine along with the arguments, the catch being that it is first necessary to break it down into two bytes (LSB and MSB) so that the machine-code routine will "understand" its number-format and be able to load it into a register pair. Although this can be done in BASIC, it's messy and slow; other ways of passing arguments to machine-code will be considered later in the chapter.

Listing 9.1

```
7D00            00100          ORG     32000
7D00 210F7D     00110          LD      HL,ARG1
7D03 3A107D     00120          LD      A,(ARG2)
7D06 17         00130          RLA
7D07 17         00140          RLA
7D08 17         00150          RLA
7D09 17         00160          RLA
7D0A B6         00170          OR      (HL)
7D0B 0600       00180          LD      B,0
7D0D 4F         00190          LD      C,A
7D0E C9         00200          RET
7D0F 00         00210 ARG1     DEFB    0
7D10 00         00220 ARG2     DEFB    0
0000            00230          END
00000 TOTAL ERRORS
ARG1    7D0F
ARG2    7D10
```

Here is the BASIC program, which operates in conjunction with the program in Listing 9.1:

```
10      LET ARG1=12
20      LET ARG2=7
30      GOSUB 100
40      POKE 28000,X
50      STOP
100     POKE 32015,ARG1
110     POKE 32016,ARG2
120     LET X=USR 32000
130     RETURN
```

The BASIC subroutine at line 100 will POKE each pair of values into the area set aside for storage in the machine-code routine by the DEFB pseudo-ops. Execution of the USR call will transfer machine execution to the machine-code routine; this will put the ARG2 value in the A register, shift it left four times to get it into the correct bit positions, then OR the ARG1 value into the four free bit locations in A. The result is then transferred to C, and B is zeroed to make sure that there are no extraneous values in it. On RETurn to BASIC, the value in the BC pair is transferred to the variable X, and on RETURN to line 40, this value is POKEd to the storage area.

Retrieval of the values from storage requires a similar pair of programs. In the machine-code routines, AND would be used instead of OR, and right-shifts instead of left-shifts. Only one argument would have to be passed to the machine-code program, which would be the byte value extracted from storage by a PEEK, but a problem arises with the results; there are two resulting arguments and the only way of transferring a result back to BASIC is to put it in the BC register pair. We could put one result in each register, but then we have to separate them out again in BASIC (dividing by 256 and noting the result and the remainder).

This example illustrates both the strengths and the weaknesses of hybrid programming with USR. Most of the coding that is being used is taken up with passing arguments back and forth; perhaps this is an extreme example of the problem, but in general, the programmer will need a more convenient way of transferring multiple arguments between BASIC and machine code if he is to use the technique frequently. The answer to the problem is to develop ways of reading arguments out of BASIC programs directly, and of "zapping" the results back into a BASIC program listing. In order to do this, we will need to understand how to use assembly language to scan a BASIC listing in memory when searching for an item of data, and how to modify a BASIC program in memory so that it will execute differently (but without crashing!) when subsequently run under control of the ZX81 ROM.

These techniques will be covered later. First, a final example of hybrid programming which takes us back to the problems of slow execution encountered in the chapter on Supergraphics. The difficulty was that we could not define a multi-line graphics image in a single string, except by including in the string a large number of blank spaces which would erase other images on the screen as the first image moved around.

We used instead a set of string-literals or variables, one for each line of the image, but this gave the BASIC interpreter more work to do and made the program run a little too slowly. Once we have decided to venture into machine code to fix the problem, several solutions suggest themselves. The byte values representing the graphics image can be stored somewhere in memory and sent to the screen either by loading to the Display File, using RST 10H, or using PRINT$. Special byte values can be interspersed among these strings, signifying a carriage return or a PRINT AT position,

and the machine-code routine can be designed to execute these as it encounters them in reading through the strings. The particular solution suggested here is put forward, in preference to other techniques, only because it requires the minimum of machine-language coding and leaves as much as possible to BASIC. In particular, the strings making up the image are all in the BASIC program and this speeds up the editing process that is likely to be necessary during development of graphics-based programs.

Listing 9.2

```
1000 SLOW
1001 LET A$="     "
1002 LET M=32000
1003 LET F=4
1004 LET S=2000
1005 FOR C=5 TO 8
1010 PRINT AT F,C;
1020 PRINT "▀▀▀▀▀";
1030 RAND USR M
1040 PRINT "▀▀▀▀▀";
1050 RAND USR M
1060 PRINT "▀▀▀▀▀";
1065 GOSUB S
1070 NEXT C
1100 STOP
2000 PRINT AT F,C;
2010 PRINT A$;
2020 RAND USR M
2030 PRINT A$;
2040 RAND USR M
2050 PRINT A$;
2060 RETURN
```

Listing 9.2 shows a short BASIC program to print an image consisting of three lines, each five characters long, and move it across the screen while erasing previous images. The program could probably be speeded up further; for example, the erasing is done with three lines of five blanks, but in fact only the first character in each line needs to be erased, since the succeeding image will itself overwrite the rest of it.

The point of the program, however, is to show how the cursor position can be moved around the screen by a short machine-code program instead of using the BASIC PRINT AT statement. The machine-code program must be tailored to the requirements of the BASIC program and in this case we need to move the cursor down one line and five spaces left, ready to print the next five characters, after each PRINT statement has been executed. The machine-code assembly is shown in Listing 9.3, which also serves to illustrate use of the PRNTAT routine in ROM.

In order to calculate the correct adjustments to make to the PRINT-position, we have to realise that the ZX81 stores information about the current cursor location in different ways. The values used for column positions in the BASIC PRINT AT statement, for example, vary from zero to 31 from left to right across the screen. The current value for the column position is stored by

Listing 9.3

```
7D00            00100         ORG     32000
08F5            00110 PRNTAT  EQU     08F5H
403A            00120 ROW     EQU     16442
4039            00130 COL     EQU     16441
7D00 213A40     00140         LD      HL,ROW
7D03 7E         00150         LD      A,(HL)
7D04 47         00160         LD      B,A     ;SAVE ROW IN B
7D05 3E18       00170         LD      A,24
7D07 90         00180         SUB     B       ;RESULT IN A IS BASIC ROW NUMBER
7D08 3C         00190         INC     A       ;NEXT LINE DOWN
7D09 47         00200         LD      B,A     ;ONE PRNTAT PARAMETER
7D0A 213940     00210         LD      HL,COL
7D0D 7E         00220         LD      A,(HL)
7D0E 57         00230         LD      D,A     ;SAVE COLUMN IN D
7D0F 3E1C       00240         LD      A,28
7D11 92         00250         SUB     D       ;RESULT IN A IS BASIC COLUMN NUMBER LESS 5
7D12 4F         00260         LD      C,A     ;OTHER PRNTAT PARAMETER
7D13 CDF508     00270         CALL    PRNTAT
7D16 C9         00280         RET
0000            00290         END
00000 TOTAL ERRORS
COL     4039
PRNTAT  08F5
ROW     403A
```

BASIC in location 16441, but it's the other way round; the value decreases from 31 to zero as we go from left to right across the screen. However, when working out a column parameter for the ROM PRNTAT routine, we have to use the BASIC convention for the column value, which is loaded into the C register before CALLing the ROM routine. Similarly with the row number, a value from zero to 21 is loaded into the B register as a parameter for PRNTAT, and corresponding to the BASIC convention; the value is stored "in reverse" in location 16442.

To find out the current cursor location on the screen immediately on departure from BASIC execution *via* a USR call, we can just read 16441/2, and this is done in lines 150 and 220 in Listing 9.3. However, we can't change the cursor location just by loading new values into these addresses; we have to use PRNTAT. The program therefore converts the values representing the cursor location from one convention to the other. Line 180 creates the row number as PRNTAT understands it, and the next line increases the row number since we want to print on the next line down. Line 250 does the same for the column, but the value 28 in line 240 allows for the fact that we want to move the cursor five places left because the image is five characters long. A one-character image, for example, would require a value of 32 in line 240.

Having placed the parameters in the B and C registers, we can CALL PRNTAT and RETurn to BASIC, which will print the next character-string on the screen in the correct position. The effect

of the machine-code routine is therefore similar to that of a specialised cursor-control code embedded in the BASIC program, moving the cursor location in just the right manner for this particular image.

There are some disadvantages to this technique. If the image is accidently sent too far to the left or too far down by the BASIC program, for example, the parameters sent to PRNTAT will be invalid. However, this shouldn't cause a system crash; the program will terminate with a B error code. The machine-code routine is not a general-purpose utility; this would require two separate routines, one for moving the cursor down and one for moving it to the left. Probably one would arrange for BASIC to POKE an argument into the LEFT routine, specifying the number of spaces that the cursor position should be moved to the left.

The graphics of the BASIC program are speeded up by the procedure considerably, and the remaining delay is largely attributable to the FOR/NEXT loop. Any idea of replacing this with a short machine-code routine should be carefully considered before any further amendment is made to the program. It really involves changing the whole structure of the program, rather than just one part of it, and it would probably be quicker in the long run to write the whole thing in Assembler, which is the logical next step.

There is, however, one variation on the technique described above which makes the BASIC program execute a little faster because it gives the interpreter less work to do. Two changes have to be made to Listing 9.3: line 240 has to be changed to read:

```
        LD    A,27
```

so that the cursor location is moved one space left, and a new line 275 is inserted:

```
275     LD    BC,0
```

Now we can print a complete screen-image with a single BASIC line such as:

```
100     PRINT "AAAAA";CHR$ USR M;"BBBBB";CHR$ USR M;"CCCCC";
```

What happens is that, on return from the machine-code routine, the interpreter is instructed to print the contents of the BC register pair. However, we have loaded BC with zero, and the CHR$ equivalent of this is a blank space. The space is dumped to the screen without being noticed (assuming we are displaying on a white background), and this moves the cursor back to the position at which we want to print the next line of the screen image. The single BASIC line gives the interpreter much less work to do than the sequence of lines 1020-1060 in Listing 9.2, and the routine executes faster accordingly.

READING AND WRITING TAPES

All sorts of applications can be envisaged in which you might want to be able to read a tape from cassette into memory, using the BYTIN routine given earlier. In practice, however, these will be divided into reading BASIC tapes and reading tapes created in your own format. An example of the first application would be a project to write a program to discover the name of a tape, where the name is not quoted in the program listing.

This breaks down into two main tasks: store the name in a buffer while you are reading it in, and note when the last character of the name has been read in. The procedure for storing values in a buffer can be seen in Listing 8.2, although this application will be much more straightforward than GETSTG, and merely involves transferring values from the C register to successive memory locations in the buffer. Alternatively, you could dump them to the screen as they come in; the values are the codes for the characters in the Sinclair character set and can be printed using RST 10H without further processing.

Let's assume that you have a subroutine called BUFSTR to store the characters in a buffer. How do you know when you have come to the end of the name? The last character is "inverted", which means in practice that it is the only one with bit 7 set; that is why the user is not allowed to use inverted characters when typing in the name of a tape. You will be CALLing BYTIN to read in each byte and could test bit 7 with:

```
BIT 7,C
```

However, why not impress your friends by using a more elegant technique? You can do it this way:

```
LOOP   CALL BYTIN
       CALL BUFSTR
       JP   P,LOOP
       NOP          ;program continues
```

The P/V flag will be set if bit 7 is set; we're not carrying out a parity or overflow check, as it happens, but this is not the point. As mentioned earlier, the literal definition of the action caused by a Z80 instruction is what matters rather than its intended use. Remember that BUFSTR should not disturb the condition of the P/V flag.

Perhaps you don't care what the name is but want to extract some information from the tape, such as the variable table, ignoring the program listing, Display File and the record of the low RAM values at the time the tape was made. In practice you will have to read in the values from the low RAM, keeping track of the number of bytes you have read in so far, until you know that you have arrived at the two bytes representing the values in locations

16400 and 16401 at the time that the tape was made. These tell you the address of the start of the variable table, in LSB/MSB format. Why not pass them into the HL pair for storage, PUSHing and POPping as appropriate to preserve their values? That's fine, but don't forget that the first byte goes into L and the second into H! This is such an obvious point, but it catches people every time.

Now you can load 16401 into BC and subtract it from HL. The result is the total number of bytes to be read in from tape, from the current position in your program onwards, to reach the first byte of the variables on the tape. You may want to look at page 178 of the Sinclair manual if you have trouble picturing what is going on here. Any other section of the tape, such as the Display File, can be isolated in the same way. Perhaps, if you are reading in the variables, you are writing them to the end of the Display File in memory, erasing the previous variable table in memory. This should work out all right; to find out when you have reached the end of the variable table on tape, you need to subtract the value that would have been loaded into 16400/1 from the value that would have been loaded into 16404/5 (start of free memory), so these values had better be read off the tape as well, and can be used to update the low RAM so that it corresponds to the new version of the memory map that you have created.

Reading tapes made in your own format is much more straightforward; you will have created the format to make things easy for yourself. For example, if you have a routine to dump sections of memory to tape using BYTOUT, and want to be able to read them back into the same locations, there are two pieces of information you need on loading: the address to load the first byte, and the length. These values can be sent to tape first of all (four CALLs to BYTOUT), after which the core dump is executed by incrementing HL to point to each memory location in turn and CALLing BYTOUT each time. BYTOUT always dumps the contents of the memory location pointed to by HL, so you can either put the first four values in memory immediately preceding the addresses to be dumped, or else use a tiny buffer, put the values there and make HL point to it.

On loading, the first four bytes can be transferred to two register pairs, probaboy HL and BC, and saved on the stack. Loading the remaining bytes on the tape will then involve incrementing HL while decrementing BC to zero.

SEARCHING IN-MEMORY BASIC PROGRAMS

Earlier I complained about the difficulty of implementing certain types of hybrid program on the ZX81, owing to the awkwardness of passing arguments to machine-code routines. The answer is to make the machine code do the work, by collecting the arguments from the BASIC program. This eliminates the need for any POKEs, but makes it necessary for the machine-code routine to be able to scan through the BASIC program looking for an identifying marker-value which will signify "argument follows", after which the argument can be collected and processed.

In practice there are two different kinds of problem. If the argument you want to collect is "fixed", it can be stored in the BASIC program listing; if it is "variable" in the sense that it changes or is determined during program execution, then it is going to be somewhere in the variable table, and that is where you have to look.

An example of the second kind of problem would be the task of scanning through the variable table looking for the current values of ARG1 and ARG2, as used in Listing 9.1 and the associated BASIC coding earlier in this chapter. You have to find the start of the variable table from the values stored at 16400/01, look at the name of the first variable, skip to the next if it is not the one you want, and so on. The different types of variable have different lengths in the table and it is tedious, but fairly straightforward, to work out the logic of this skipping process. The information necessary to do this is all contained on pages 172-174 of the Sinclair manual; the only thing that might lead you astray is the misprint on page 173. In the first byte that is used to store the control variables of a FOR/NEXT loop, the value in bits 4-0 will be the character code of the control variable-name minus 20H, not just the character code.

Working out a routine of this sort gives you the equivalent of the VARPTR statement available in more sophisticated BASICs, which will return the address at which a specified variable is stored. Values computed by the BASIC program and not shown explicitly anywhere in the listing will then be available for inspection. There is, however, a catch; all the numeric variables will be in five-byte floating-point format! You could consider writing a routine to convert from this format to binary so that the values can be passed to your machine-code program, but I would advise against it. The floating-point routines are the longest and most complex in the ROM; it is better to make the ROM do the work of conversion. How is this accomplished?

The answer seems to be to use STR$. By using a line like

```
100    LET A$=STR$ ARG1
```

you can compel the interpreter to work out the decimal representation of the current value of ARG and store it as a string of decimal numbers, in the variable table, as the variable A$. This makes it possible to pick it up from the table, and send it to the DECBIN routine described in Chapter 8, after which it will be in the right format for a machine-code program to operate on.

Collecting an argument from a BASIC listing requires a different approach. It will often not be necessary to scan the whole program; instead, one can examine the value held at 16406/7 at the time that the USR call is executed. This value will be the memory address in the BASIC program containing the character or BASIC token that has just been executed. Consider the following couple

of lines:

```
100   LET X=USR OR
200   REM 10,128
```

This might be used to pass two values to a machine-code program, at an address equated with the variable OR, which would perform a logical-OR on them and pass back the result into the BASIC variable X. When control passes to the machine-code routine, locations 16406/7 hold a memory address which is somewhere within line 100 of the BASIC listing. What we want to do is scan the listing from this point onwards until we find the two arguments in the REM line. First we get past line 100 to the beginning of line 200, using some coding such as the following:

```
        LD    HL,(16406)
LOOP1   LD    A,(HL)
        CP    118
        JR    Z,CONT
        INC   HL
        JR    LOOP1
```

On each iteration of the loop, the routine will examine a byte in memory to see if it is the NEWLINE marker signifying the end of a BASIC line. If not, it examines the next byte; otherwise it jumps to the CONTinuation of the program.

Having found the end of one line, the idea is to skip the next four bytes which contain the number and length of the line and examine the following byte. In the case of the two-line BASIC program fragment quoted above, the value in this byte will be 234 decimal which is the token for REM. The routine can check that this is so, then take values from the following bytes which represent the arguments.

However, it depends how many failsafes you want to build in; suppose the programmer has failed to put a REM line as the line following his USR call? He may have thoughtlessly interspersed some other line between the two, in which case you still want to find his REM line, or he may have put no REMs in the program at all, in which case you want to avoid a system crash as the routine fruitlessly searches through to the end of the program and beyond. Thinking ahead in this way, and anticipating user errors, makes all the difference between a flexible and reliable machine-code utility, and one which is so demanding and dangerous that even the author finds himself avoiding the use of it.

The following coding fragment will take care of both the contingencies mentioned above; it can be CALLed from the point at

which the routine given above has exited the loop and jumped to CONT:

```
        INC     HL          ;point to first line number byte
LOOP2   INC     HL          ;second byte
        INC     HL          ;first byte of line length
        LD      E,(HL)
        INC     HL
        LD      D,(HL)      ;now line length in DE
        INC     HL          ;first byte of line text
        LD      A,(HL)
        CP      234         ;is it REM token?
        RET     Z           ;yes, return
        ADD     HL,DE       ;no, so add line length to line
                             start address to give start address
                             of next line in HL
        LD      A,(HL)
        CP      118         ;is it another NEWLINE, meaning end
                             of BASIC program text?
        JR      Z,ERROR4    ;if so jump to error-handling
                             routine for "no REMs"
        JR      LOOP2       ;else go back and study the next line
```

There are all sorts of ways in which you may want to study BASIC in-memory listings and it isn't possible to give details of all the contingencies, but the coding shown above should illustrate the general approach. There are, of course, cases in which you would need to read through the whole BASIC program from location 16509, the most obvious example being the implementation of READ, DATA and RESTORE. No one has got around to providing this extension to ZX81 BASIC yet, although it is certainly needed. Data items would probably be stored in REM lines like this:

```
100    REM DATA ALPHA,45,BETA
```

and access to the READ utility would require a pair of lines like this:

```
1000   RAND USR READ
1010   REM A$,X,B$
```

Working out what the utility would need to do allows us to see what unsolved problems lie in our path. We already have a routine for picking out the address at which the arguments can be found - in this case they are three BASIC variables in which data values have to be stored. The utility will be able to differentiate

between string and numeric variables by examining memory to see if a character is followed by the "$" character. Data items are read sequentially as they occur in a BASIC listing, no matter where they are, so the utility needs to retain an address in storage representing the point that it reached on the last occasion that it was scanning the BASIC program, and the next READ statement will cause it to search for the next data item after that point; it will have to look at each line commencing with a REM and check whether it is followed by the characters denoting "DATA".

So far so good, but how can we get the characters in the REM line, following the word DATA, into the variable table? We will have to scan the table for each variable to see if it exists yet; if so, it is overwritten with the new value, otherwise a new variable is set up at the end of the table. In the case of string variables, the length of the string in the DATA statement must be compared with the length of the string currently equated with the same variable; this information is stored in the variable table along with other details about string variables. Then the variables following this variable will have to be moved up or down in memory, to create the right amount of storage for the new-length string, and locations 16404/5 will have to be updated accordingly. Fortunately, there will be no material in the locations following the variable table that needs to be preserved; this space is used during editing and during BASIC program execution but not during USR routines.

Unfortunately we run up against a snag at the last minute; we have no routine for converting decimal numeric values into five-byte floating-point format, and as mentioned earlier, even hardened assembly-language programmers are liable to flinch at the prospect of writing such a routine. Without it, we are unable to fill the five bytes, reserved in the variable table for numeric values, with the correct values to represent a number such as 45 in the example given above. A project to write such a routine would probably quadruple the length of the complete READ/DATA utility. It seems better to make the ROM do the work, and ask the user to read all data items into string variables only. This will in fact simplify and speed up the operation of the utility, and in order to convert a string value such as "45" into the numeric value 45, the user only has to write a line such as:

```
1000   LET X=VAL X$
```

which will make the ROM do the conversion during program execution. The reader may of course feel quite happy about writing his own floating-point conversion routine and getting it right (remember, the authors of the ROM got it wrong the first time). The point of the account given here of the planning that should be done before starting coding, is to show how the fundamental details of the working of a routine can't be settled until the programmer has decided how all stages of his task will be implemented.

Modifying BASIC programs is, from the assembly-language programmer's point of view, little different from the job of searching

through BASIC listings. Generally it will be necessary to scan through the listing to find the area to be modified, after which the necessary values can be loaded in to overwrite the existing text. This is already a widespread practice, as it is used to load the values representing a machine-code program into a REM line set up to contain the correct number of dummy characters. Generally the REM line is the first in the program, as the programmer then knows that the first memory location to be altered is 16514. As we can see from the material discussed in the last two chapters, however, there's no real need to restrict oneself to this practice. Any line can be overwritten provided that it is preceded by some sort of unique identifying token or tokens, such as the character-string "DATA", which is known not to occur elsewhere in the program, and which will allow a program-scanning subroutine to locate the line to be modified.

Nor is it necessary to type in hundreds of dummy characters to be overwritten. A less tedious procedure is to create the entire BASIC line. If it's to occur in the middle of a listing, the subsequent lines and Display File must be LDIR'red to higher memory locations to make room for it; the chosen line number is then stored (MSB/LSB format) so that it occurs immediately after the NEWLINE at the end of the preceding BASIC line, and is followed by the length of the line (LSB/MSB). The required values are then loaded into successive memory locations, and terminated by a NEWLINE, which will be followed by the first byte of the remainder of the BASIC program. The only updating that is then required is to note the new values to be loaded to 16396/7 (start of Display File) and 16400/1 (start of variables). The value for 16404/5 (start of free memory) will just be one more than start-of-variables, assuming that the programmer doesn't mind wiping out any existing variable table.

CONCLUSION

The programmer may baulk at some of the projects suggested in this chapter, which seem very ambitious. On the other hand, much more sophisticated programming aids and other facilities are available for other computers than have yet appeared for the ZX81, and there is no obvious reason for this. Probably many professional programmers have avoided the machine because of the obstacles it presents to program development or in the expectation that it would become less popular with users. At the time of writing it seems clear that there will be at least a million ZX81's worldwide by the end of 1982, and it doesn't seem likely that the machine will "go away", or be superseded by later models such as the Spectrum. On the contrary, a wide range of software is now demanded by users, including BASIC compilers, additional languages, BASIC enhancements and utilities such as object code relocaters and better assemblers, as well as more sophisticated games and other leisure programs. Perhaps the information supplied in the present book will encourage some readers to set about the task of meeting this demand.